Cray XT™ Series Programming Environment User's Guide

S–2396–20

New Features

Cray XT™ Series Programming Environment User's Guide S–2396–20

Cross compiler platform

Added support of a standalone, cross compiler machine for creating executables to be run on Cray XT series systems (see Section 1.1, page 1).

ALPS — Added support of ALPS (Application Level Placement Scheduler). ALPS is the application launcher for CNL applications. For further information, see Section 1.2, page 1.

Create node lists by compute node attributes

Added support of the `cnselect` command. You can use `cnselect` to get a candidate list of compute nodes based on node attributes you specify. You can then use this list to launch applications on compute nodes with those characteristics. For further information see Section 1.2, page 1.

Target architecture

The target architecture (CNL or Catamount) is set automatically at log in. For further information, see Section 2.2, page 9.

IRT — Added IRT (Iterative Refinement Toolkit) to Cray XT-LibSci. You can use IRT as an efficient alternative to standard LAPACK or ScaLAPACK linear equation solvers. For further information, see Section 3.2, page 13.

ACML changes

The ACML module is no longer loaded as part of the default `PrgEnv` environment. For further information, see Section 3.3, page 16.

PETSc — Added support of PETSc (Portable, Extensible Toolkit for Scientific Computation). For further information, see Section 3.5, page 18.

OpenMP — Added support of OpenMP for PGI, PathScale, and GCC applications that are run on CNL compute nodes. For further information, see Section 3.8, page 22.

CNL — Added support of CNL. CNL is a compute node operating system; sites can use it as an alternative to Catamount. For further information, see Chapter 4, page 23.

Unsupported PGI compiler command options

Added note that the PGI `-mprof=mpi`, `-Mmpi`, and `-Mscalapack` options are not supported on Cray XT series systems (see Section 4.1.1.5, page 25).

Suppressing vectorization

Documented methods of suppressing vectorization in PGI applications (see Section 4.1.1.6, page 25).

Lustre required for CNL applications

In CNL, only I/O to Lustre file systems is supported (see Section 4.2.2, page 27.

Resolving copy-on-write problems

Modified the Portals kernel to perform a partial copy of pages when a process forks a child. The standard Linux `fork()` copy-on-write process can adversely affect Portals data transfers (see Section 4.2.11, page 29).

Creating CNL or Catamount executables

Added modules that enable you to create CNL or Catamount executables, regardless of the operating system running on the compute nodes. For further information, see Section 5.1, page 39.

PGI compilers

Documented PGI Cluster Development Kit (CDK) options not supported on Cray XT series systems. For further information, see Section 5.2.1, page 40.

GNU Fortran 95 compiler

Added support of the GNU Fortran 95 compiler. For further information, see Section 5.2.2, page 42.

PathScale compilers

Added support of the PathScale C, C++, and Fortran compilers. For further information, see Section 5.2.3, page 43.

Methods for getting node status

Added the `xtprocadmin -A` command, which generates a report showing node attributes. Also enhanced the `xtshowmesh` and `xtshowcabs` reports. For further information, see Chapter 6, page 47.

PBS Pro `-l` *resource_type* options

Documented changes in PBS Pro resource-type specifications (such as `-l mppwidth` replacing `-l size` (see Section 9.2, page 68).

Trace reports about memory allocation and deallocation

Added the `-tracemalloc` option to the `yod` command to generate trace diagnostics for `malloc()` and `free()` calls (see Section 10.1, page 73).

CrayPat sampling

Added support of CrayPat sampling (asynchronous) experiments (see Section 11.2.1, page 86).

Cray Apprentice2 desktop

Added support of Cray Apprentice2 running on a standalone Linux based machine (see Section 11.3, page 88).

Rank placement method for CNL applications

Added support of the `yod` placement method (rank-sequential order) for CNL applications (see Section 12.2.2, page 93).

Record of Revision

Version	*Description*
1.0	December 2004 Draft documentation to support Cray XT3 early-production systems.
1.0	March 2005 Draft documentation to support Cray XT3 limited-availability systems.
1.1	June 2005 Supports Cray XT3 systems running the Cray XT3 Programming Environment 1.1 and UNICOS/lc 1.1 releases.
1.2	August 2005 Supports Cray XT3 systems running the Cray XT3 Programming Environment 1.2 and UNICOS/lc 1.2 releases.
1.3	November 2005 Supports Cray XT3 systems running the Cray XT3 Programming Environment 1.3 and UNICOS/lc 1.3 releases.
1.4	April 2006 Supports Cray XT3 systems running the Cray XT3 Programming Environment 1.4 and UNICOS/lc 1.4 releases.
1.5	August 2006 Supports limited availability (LA) release of Cray XT series systems running the Cray XT series Programming Environment 1.5 and UNICOS/lc1.5 releases.
1.5	November 2006 Supports general availability (GA) release of Cray XT series systems running the Cray XT series Programming Environment 1.5 and UNICOS/lc 1.5 releases.
2.0	June 2007 Supports limited availability (LA) release of Cray XT series systems running the Cray XT series Programming Environment 2.0 and UNICOS/lc 2.0 releases.
2.0	October 2007 Supports general availability (GA) release of Cray XT series systems running the Cray XT series Programming Environment 2.0 and UNICOS/lc 2.0 releases.

Contents

Page

Page

Page

Page

Page

Figures

Tables

Page

Preface

The information in this preface is common to Cray documentation provided with this software release.

Accessing Product Documentation

With each software release, Cray provides books and man pages, and in some cases, third-party documentation. These documents are provided in the following ways:

CrayDoc

The Cray documentation delivery system that allows you to quickly access and search Cray books, man pages, and in some cases, third-party documentation. Access this HTML and PDF documentation via CrayDoc at the following locations:

- The local network location defined by your system administrator
- The CrayDoc public website: `docs.cray.com`

Man pages

Access man pages by entering the `man` command followed by the name of the man page. For more information about man pages, see the `man`(1) man page by entering:

```
% man man
```

Third-party documentation

Access third-party documentation not provided through CrayDoc according to the information provided with the product.

Conventions

These conventions are used throughout Cray documentation:

Convention	Meaning
`command`	This fixed-space font denotes literal items, such as file names, pathnames, man page names, command names, and programming language elements.
variable	Italic typeface indicates an element that you will replace with a specific value. For instance, you may replace *filename* with the name `datafile` in your program. It also denotes a word or concept being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a syntax representation for a command, library routine, system call, and so on.
. . .	Ellipses indicate that a preceding element can be repeated.
`name`(N)	Denotes man pages that provide system and programming reference information. Each man page is referred to by its name followed by a section number in parentheses. Enter: `% man man` to see the meaning of each section number for your particular system.

Reader Comments

Contact us with any comments that will help us to improve the accuracy and usability of this document. Be sure to include the title and number of the document with your comments. We value your comments and will respond to them promptly. Contact us in any of the following ways:

E-mail:
`docs@cray.com`

Telephone (inside U.S., Canada):
1–800–950–2729 (Cray Customer Support Center)

Telephone (outside U.S., Canada):
+1–715–726–4993 (Cray Customer Support Center)

Mail:
Customer Documentation
Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120–1128
USA

Cray User Group

The Cray User Group (CUG) is an independent, volunteer-organized international corporation of member organizations that own or use Cray Inc. computer systems. CUG facilitates information exchange among users of Cray systems through technical papers, platform-specific e-mail lists, workshops, and conferences. CUG memberships are by site and include a significant percentage of Cray computer installations worldwide. For more information, contact your Cray site analyst or visit the CUG website at `www.cug.org`.

Introduction [1]

This guide describes the Cray XT series Programming Environment products and related application development tools. In addition, it includes procedures and examples that show you how to set up your user environment and build and run optimized applications. The intended audience is application programmers and users of Cray XT series systems. Prerequisite knowledge is a familiarity with the topics in the *Cray XT Series System Overview*. For information about managing system resources, system administrators can see the *Cray XT Series System Management* manual.

Note: Functionality marked as deferred in this documentation is planned to be implemented in a later release.

1.1 The Cray XT Series System Environment

The system on which you run your Cray XT series applications is an integrated set of Cray XT series compute node and service node components. You log in either to a service node or a standalone cross-compiler machine and use the Cray XT series Programming Environment and related products to create your executables. You run your executables on Cray XT series compute nodes.

The operating system is UNICOS/lc; it has compute node and service node components. Compute nodes run either the CNL or the Catamount operating system. Service nodes run SUSE LINUX. For details about the differences between CNL and Catamount, see Chapter 4, page 23.

1.2 The Cray XT Series Programming Environment

The Cray XT series Programming Environment includes the following products and services:

- PGI compilers for C, C++, and Fortran (see Chapter 5, page 39).
- GNU compilers for C, C++, and Fortran (see Chapter 5, page 39).
- PathScale compilers for C, C++, and Fortran (see Section 5.2.3, page 43).
- Parallel programming models:
  - Cray MPICH2, the Message Passing Interface routines (see Section 3.6, page 18).

  - Cray SHMEM shared memory access routines (see Section 3.7, page 20).
  - OpenMP shared memory model routines, Fortran directives, and C and C++ pragmas (see Section 3.8, page 22). OpenMP is not supported for applications running under Catamount.

- Cray XT-LibSci scientific library, which includes:
  - Basic Linear Algebra Subprograms (BLAS)
  - Linear Algebra (LAPACK) routines
  - ScaLAPACK routines
  - Basic Linear Algebra Communication Subprograms (BLACS)
  - Iterative Refinement Toolkit (IRT)
  - SuperLU routines

  For further information about Cray XT-LibSci, see Section 3.2, page 13.
- AMD Core Math Library (ACML), which includes:
  - Fast Fourier Transform (FFT) routines
  - Math transcendental library routines
  - Random number generators
  - GNU Fortran libraries

  For further information about ACML, see Section 3.3, page 16.
- PETSc (Portable, Extensible Toolkit for Scientific Computation). For further information, see Section 3.5, page 18.
- FFTW (see Section 3.4, page 17)
- A subset of the glibc GNU C Library routines for compute node applications (see Section 3.1, page 13).
- The Performance API (PAPI) (see Section 11.1, page 83).

In addition to Programming Environment products, the Cray XT series system provides these application development products and functions:

- The Application Level Placement Scheduler (ALPS) utility for launching applications on CNL compute nodes (`aprun` command), killing processes (`apkill` command), and getting status about applications (`apstat` command). See Chapter 7, page 53 for a description of `aprun` and Appendix E, page 201 for a description of common ALPS error messages.
- The `yod` command for launching applications on Catamount compute nodes (see Chapter 8, page 59).
- The `cnselect` command for generating a candidate list of compute nodes based on user-specified selection criteria; you can use this list on `aprun -L` *nodes* or `yod -list` *processor-list* commands to launch an application on compute nodes with those characteristics (see the `cnselect`(1) man page).
- Lustre parallel file system (see Section 2.4, page 11).
- The `xtprocadmin -A` command for generating a report showing the attributes of the compute nodes (see Chapter 6, page 47).
- The `xtshowmesh` and `xtshowcabs` commands for generating reports showing the status of compute nodes (see Chapter 6, page 47).

The following optional products are available for Cray XT series systems:

- PBS Pro batch processing system (see Chapter 9, page 67).

  Note: If your site has installed another batch system, please contact the appropriate vendor for the necessary installation, configuration, and administration information. For example, contact Cluster Resources, Inc. (`http://www.clusterresources.com/`) for documentation specific to Moab products.

- TotalView debugger (see Section 10.2, page 74). The TotalView debugger is available from TotalView Technologies, LLC (`http://www.totalviewtech.com/Documentation/`).
- GNU debugger (see Section 10.3, page 81).
- CrayPat performance analysis tools (see Section 11.2, page 84).
- Cray Apprentice2 performance visualization tool (see Section 11.3, page 88).

1.3 Documentation Included with This Release

Table 1 lists the manuals and man pages that are provided with this release. All manuals are provided as PDF files, and some are also available as HTML files. You can view the manuals and man pages through the CrayDoc interface or move the files to another location, such as your desktop.

Note: You can use the Cray XT Series System Documentation Site Map on CrayDoc to link to all Cray manuals and man pages included with this release.

Table 1. Manuals and Man Pages Included with This Release

Cray XT Series Programming Environment User's Guide (this manual)
Cray XT Series Programming Environment man pages
Cray XT Series Release Overview
Cray XT Series System Overview
PGI User's Guide
PGI Fortran Reference
PGI Tools Guide
Cray XT Series Programming Environments Installation Guide manual
Modules software package man pages
Cray MPICH2 man pages (read `intro_mpi`(3) first)
Cray SHMEM man pages (read `intro_shmem`(3) first)
AMD Core Math Library (ACML) manual
Cray XT-LibSci man pages(read `intro_libsci`(3s) first)
Iterative Refinement Toolkit man pages(read `intro_irt`(3) first)
SuperLU Users' Guide
FFT man pages (`intro_fft`(3), `intro_fftw2`(3), `intro_fftw3`(3))
PBS Pro Release Overview, Installation Guide, and Administration Addendum
PBS Pro Quick Start Guide
PBS Pro User Guide
PBS Pro External Reference Specification
TotalView `totalview`(1) *man page*

Performance API (PAPI) man pages

Using Cray Performance Analysis Tools manual

CrayPat and Cray Apprentice2 man pages (read `craypat`(1) and `app2`(1) first)

Additional sources of information:

- PGI manuals at `http://www.pgroup.com` and the `pgcc`(1), `pgCC`(1), `pgf95`(1), and `pgf77`(1) man pages available through the `man` command.
- *Using the GNU Compiler Collection (GCC)* manual at `http://gcc.gnu.org/` and the `gcc`(1), `g++`(1), `gfortran`(1), and `g77`(1) man pages available through the `man` command.
- *QLogic PathScale Compiler Suite User's Guide* at `http://www.pathscale.com/docs/html` and the `pathcc`(1), `pathCC`(1), `pathf95`(1), and `eko`(7) man pages available through the `man` command.
- MPICH2 documents at `http://www-unix.mcs.anl.gov/mpi/mpich2/` and `http://www.mpi-forum.org`.
- OpenMP documents at `http://www.openmp.org`.
- The *ScaLAPACK Users' Guide* at `http://www.netlib.org/scalapack/slug/`.
- SuperLU documents at `http://crd.lbl.gov/~xiaoye/SuperLU/`.
- PETSc documents at `http://www-unix.mcs.anl.gov/petsc/petsc-as`.
- FFTW documents at `http://www.fftw.org/`.
- PAPI documents at `http://icl.cs.utk.edu/papi/`.
- Lustre documentation (`http://manual.lustre.org/`).
- SUSE LINUX man pages available through the `man` command.

Setting Up the User Environment [2]

Configuring your user environment on a Cray XT series system is similar to configuring a typical Linux workstation. However, there are steps specific to Cray XT series systems that you must take before you begin developing applications.

2.1 Setting Up a Secure Shell

Cray XT series systems use `ssh` and `ssh`-enabled applications such as `scp` for secure, password-free remote access to the login nodes.

Before you can use the `ssh` commands, you must generate an RSA authentication key. The process for generating the key depends on the authentication method you use. There are two methods of passwordless authentication: with or without a passphrase. Although both methods are described here, you must use the latter method to access the compute nodes through a script or when using a system monitor command such as `xtps`.

For more information about setting up and using a secure shell, see the `ssh`(1), `ssh-keygen`(1), `ssh-agent`(1), `ssh-add`(1), and `scp`(1) man pages. For further information about system monitor commands, see the *Cray XT Series System Management* manual.

2.1.1 RSA Authentication with a Passphrase

To enable ssh with a passphrase, complete the following steps.

1. Create a $HOME/.ssh directory and set permissions so that only the file's owner can access them:

   ```
   % mkdir $HOME/.ssh

   % chmod 700 $HOME/.ssh
   ```

2. Generate the RSA keys by using the following command:

   ```
   % ssh-keygen -t rsa
   ```

   and follow the prompts. You will be asked to supply a passphrase.

3. The public key is stored in your $HOME/.ssh directory. Use the following command to copy the key to your home directory on the remote host(s):

   ```
   % scp $HOME/.ssh/key_filename.pub \

   username@system_name:.ssh/authorized_keys
   ```

   Connect to the remote host by typing the following commands.

   If you are using a C shell, use:

   ```
   % eval `ssh-agent`

   % ssh-add
   ```

   If you are using a Bourne shell, use:

   ```
   $ eval `ssh-agent -s`

   $ ssh-add
   ```

   Type your passphrase when prompted, followed by:

   ```
   % ssh remote_host_name
   ```

2.1.2 RSA Authentication without a Passphrase

To enable ssh without a passphrase, complete the following steps.

1. Create a $HOME/.ssh directory and set permissions so that only the owner of the file can access them:

   ```
   % mkdir $HOME/.ssh
   % chmod 700 $HOME/.ssh
   ```

2. Generate the RSA keys by typing the following command:

   ```
   % ssh-keygen -t rsa -N ""
   ```

   and following the prompts.

3. The public key is stored in your $HOME/.ssh directory. Type the following command to copy the key to your home directory on the remote host(s):

   ```
   % scp $HOME/.ssh/key_filename.pub \
   username@system_name:.ssh/authorized_keys
   ```

   Note: This step is not required if your home directory is shared.

4. Connect to the remote host by typing the following command:

   ```
   % ssh remote_host_name
   ```

2.2 Using Modules

The Cray XT series system uses modules in the user environment to support multiple versions of software, such as compilers, and to create integrated software packages. As new versions of the supported software and associated man pages become available, they are added automatically to the Programming Environment, while earlier versions are retained to support legacy applications. You can use the default version of an application or Modules system commands to choose another version.

The PrgEnv module loads the Programming Environment and related product modules. To load the default PrgEnv module, use:

```
% module load PrgEnv
```

To load specific compiler suite modules, use one of the following commands:

```
% module load PrgEnv-pgi
% module load PrgEnv-gnu
% module load PrgEnv-pathscale
```

The target environment module is automatically loaded at log in. If the compute nodes are running CNL, the xtpe-target-cnl module is automatically loaded. If the compute nodes are running Catamount, the xtpe-target-catamount module is automatically loaded.

For some products, additional modules may have to be loaded. The chapters addressing those products specify the module names and the conditions under which they must be loaded.

Modules also provide a simple mechanism for updating certain environment variables, such as PATH, MANPATH, and LD_LIBRARY_PATH. In general, you should make use of the modules system rather than embedding specific directory paths into your startup files, makefiles, and scripts.

To find out what modules have been loaded, use:

The Base-opts module is loaded by default. Base-opts loads the OS modules in a versioned set that is provided with the release package.

To get a list of all available modules, use:

```
% module avail
```

To switch from one module to another, use:

```
% module swap swap_out_module swap_in_module
```

For example, if you have been using the PGI compilers and want to use the GNU compilers instead, use:

```
% module swap PrgEnv-pgi PrgEnv-gnu
```

For further information about the Module utility, see the module(1) and modulefile(4) man pages.

2.3 Modifying the PATH Variable

You may need to modify the PATH variable for your environment. Do not reinitialize the system-defined PATH. The following example shows how to modify it for a specific purpose (in this case to add $HOME/bin to the path).

If you are using csh, use:

```
% set path = ($path $HOME/bin)
```

If you are using bash, use:

```
$ export $PATH=$PATH:$HOME/bin
```

2.4 Lustre File System

Lustre is the Cray XT file system for compute node applications. To use Lustre, you must direct file operations to paths within a Lustre mount point. You can use the df -t lustre or lfs df command to locate Lustre mount points:

```
% lfs df
UUID                 1K-blocks       Used Available  Use% Mounted on
nid00011_mds_UUID    1003524776  63414492 940110284    6% /lus/nid00011[MDT:0]
ost0_UUID            1128979112 278021080 850958032   24% /lus/nid00011[OST:0]
ost1_UUID            1128979112 254976940 874002172   22% /lus/nid00011[OST:1]
ost2_UUID            1128979112 258597116 870381996   22% /lus/nid00011[OST:2]
<snip>
filesystem summary:  16934686680 4270985104 12663701576   25% /lus/nid00011
```

If your environment has not been set up to use Lustre for I/O, see your system administrator. The Lustre I/O interface is transparent to the application programmer; I/O functions are handled by the Lustre client running on the compute nodes.

If you want to create a file with a specific striping pattern, use the Lustre lfs command. Lustre file systems include Object Storage Servers (OSSs). Each OSS hosts two Object Storage Targets (OSTs), which transfer data objects that can be striped across Redundant Array of Independent Disks (RAID) storage devices.

You may choose to create a file of multiple stripes if your application requires a higher transmission rate to a single file than can be provided by a single OSS. You may also need to stripe a file if a single OST does not have enough free space to hold the entire file. For example, the command:

```
% lfs setstripe results2 1048576 1 4
```

stripes file `results2` on four OSTs, (starting with `ost1`). The stripe size is 1048576 bytes.

For further information, see the `lfs`(1) man page.

Libraries and APIs [3]

This chapter describes the libraries and APIs that are available to application developers.

3.1 C Language Run Time Library

The Cray XT series supports subsets of the GNU C library, glibc, for CNL and Catamount applications. For details on glibc for CNL, see Section 4.2.1, page 26 and Appendix A, page 181. For details on the Catamount port of glibc, see Section 4.3.1, page 30 and Appendix B, page 187.

3.2 Cray Scientific Library

The Cray XT scientific library, XT-LibSci, includes Basic Linear Algebra Subroutines (BLAS), linear algebra routines (LAPACK), parallel linear algebra routines (ScaLAPACK), Basic Linear Algebra Communication Subprograms (BLACS), the Iterative Refinement Toolkit (IRT), and the SuperLU sparse solver routines.

For additional information about XT-LibSci routines, see the scientific libraries man pages (read `intro_libsci`(3s) first).

3.2.1 BLAS and LAPACK

The BLAS and LAPACK libraries include routines from the 64-bit `libGoto` library from the University of Texas.

If you require a C interface to BLAS and LAPACK but want to use Cray XT-LibSci BLAS or LAPACK routines, you must use the Fortran interfaces.

You can access the Fortran interfaces from a C program by adding an underscore to the respective routine names and by passing arguments by reference (rather than by value in the traditional way). For example, you can call the `dgetrf()` function as follows:

```
dgetrf_(&uplo, &m, &n, a, &lda, ipiv, work, &lwork, &info);
```

Note: C programmers using the Fortran interface are advised that arrays are required to be ordered in the Fortran column-major manner.

3.2.2 ScaLAPACK and BLACS

ScaLAPACK is a distributed-memory, parallel linear algebra library. The XT-LibSci version of ScaLAPACK is modified to work more efficiently on Cray XT series compute nodes.

The BLACS library is a set of communication routines used by ScaLAPACK and the user to set up a problem and handle the communications.

The ScaLAPACK and BLACS libraries can be used in MPI and SHMEM applications. Cray XT-LibSci under CNL also supports hybrid MPI/ScaLAPACK applications, which use threaded BLAS on a compute node and MPI between nodes. To use ScaLAPACK in a hybrid application:

1. Adjust the process grid dimensions in ScaLAPACK to account for the decrease in BLACS nodes.
2. Ensure that the number of BLACS processes required is equal to the number of nodes required, **not** the number of cores.
3. Set `GOTO_NUM_THREADS` to 2 in the PBS job script used to launch the job.

Example 1: Running a ScaLAPACK application

To run a ScaLAPACK application in regular mode (that is, 1 MPI process per core) with 16 BLACS processes on a 4x4 computational grid, use the `#PBS -l mppwidth` option to specify the number of processing elements needed (16) and the `#PBS -l mppnppn` option to specify the number of processing elements per node (2).

```
#!/usr/bin/csh

#PBS -l mppwidth=16
#PBS -l mppnppn=2
cd /lus/nid00007
aprun -n 16 ./a.out
```

Example 2: Running an ScaLAPACK hybrid application

To run the same job using a hybrid application, first reduce the number of BLACS processes from 16 to 8 (by specifying either a 2x4 or possibly a 4x2 computational grid). The additional parallelism within a node is provided through use of the threaded BLAS.

In the PBS script, only those tasks actually recognized are requested. So set `mppwidth` equal to the number of nodes required (`8`) and `mppnppn` equal to the number of PEs per node (`1`).

```
#!/usr/bin/csh

#PBS -l mppwidth=8
#PBS -l mppnppn=1
cd /lus/nid00007
setenv GOTO_NUM_THREADS 2
aprun -n 8 ./a.out
```

3.2.3 Iterative Refinement Toolkit

The Iterative Refinement Toolkit (IRT) is a library of factorization routines, solvers, and tools that can be used to solve systems of linear equations more efficiently than the full-precision solvers in Cray XT-LibSci or ACML.

IRT exploits the fact that single-precision solvers can be up to twice as fast as double-precision solvers. IRT uses an iterative refinement process to obtain solutions accurate to double precision.

IRT provides two interfaces:

- Benchmarking interface. The benchmarking interface routines replace the high-level drivers of LAPACK and ScaLAPACK. The names of the benchmark API routines are identical to their LAPACK or ScaLAPACK counterparts or replace calls to successive factorization and solver routines. This allows you to use the IRT process without modifying your application.

  For example, the IRT `dgesv()` routine replaces either the LAPACK `dgesv()` routine or the LAPACK `dgetrf()` and `dgetrs()` routines. To use the benchmarking interface, set the `IRT_USE_SOLVERS` environment variable to `1`.

  Note: Use this interface with caution; calls to the LAPACK LU, QR or Cholesky routines are intercepted and IRT is used instead.

- Expert interface. The expert interface routines give you greater control of the iterative refinement process and provide details about the success or failure of the process. The format of advanced API calls is:

`call irt_`*`factorization-method_data-type_processing-mode`*`(`*`arguments`*`)`

  such as: `call irt_po_real_parallel(`*`arguments`*`)`.

For details about IRT, see the `intro_irt`(3) man page.

3.2.4 SuperLU

The SuperLU library routines solve large, sparse nonsymmetric systems of linear equations. Cray XT-LibSci SuperLU provides only the distributed-memory parallel version of SuperLU. The library is written in C but can be called from programs written in either C or Fortran.

3.3 AMD Core Math Library

The AMD Core Math Library (ACML) module is no longer loaded as part of the default PrgEnv environment. BLAS and LAPACK functionality is now provided by Cray XT-LibSci (see Section 3.2.1, page 13). However, if you need ACML for FFT functions, math functions, or random number generators, you can load the library using the `acml` module:

```
% module load acml
```

ACML includes:

- A suite of Fast Fourier Transform (FFT) routines for real and complex data
- Fast scalar, vector, and array math transcendental library routines optimized for high performance
- A comprehensive random number generator suite:
  - Five base generators plus a user-defined generator
  - 22 distribution generators
  - Multiple-stream support

ACML's internal timing facility uses the `clock()` function. If you run an application on compute nodes that uses the *plan* feature of FFTs, underlying timings will be done using the native version of `clock()`. On Catamount, `clock()` returns elapsed time. On CNL, `clock()` returns the sum of user and system CPU times.

3.4 FFTW Libraries

The Programming Environment includes versions 3.1.1 and 2.1.5 of the Fastest Fourier Transform in the West (FFTW) library. FFTW is a C subroutine library with Fortran interfaces for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, such as the discrete cosine/sine transforms). The Fast Fourier Transform algorithm is applied for many problem sizes.

To use the default FFTW library, use:

```
% module load fftw
```

To use the FFTW 3.1.1 library, use:

```
% module load fftw/3.1.1
```

To use the FFTW 2.1.5 library, use:

```
% module load fftw/2.1.5
```

Distributed-memory parallel FFTs are available only in FFTW 2.1.5.

The FFTW 3.1.1 and FFTW 2.1.5 modules cannot be loaded at the same time. You must first unload the other module, if already loaded, before loading the desired one. For example, if you have loaded the FFTW 3.1.1 library and want to use FFTW 2.1.5 instead, use:

```
% module swap fftw/3.1.1 fftw/2.1.5
```

3.5 PETSc Library

The Programming Environment supports the 2.3.3 release of the Portable, Extensible Toolkit for Scientific Computation (PETSc) library. PETSc is an open source library of sparse solvers. There are two PETSc modules:

- `petsc` for real data
- `petsc-complex` for complex data

To switch from the PETSc module for real data to the module for complex data, use:

```
% module swap petsc petsc-complex
```

For details, see the `intro_petsc`(3) man page and `http://www-unix.mcs.anl.gov/petsc/petsc-as/index.html`.

3.6 Cray MPICH2 Message Passing Library

Cray MPICH2 implements the MPI-2 standard, except for support of spawn functions. It also implements the MPI 1.2 standard, as documented by the MPI Forum in the spring 1997 release of *MPI: A Message Passing Interface Standard.*

The Cray MPICH2 message-passing libraries are implemented on top of the Portals low-level message-passing engine. The Portals interface is transparent to the application programmer.

All Cray XT compilers support MPICH2 applications. There are two versions of the MPICH2 library available for users of the PGI or PathScale Fortran compilers. One version supports applications where the data size for the Fortran default types integer, real, and logical is 32 bits, and the other version supports applications where the data size is 64 bits. For further details, see Section 4.1.1.1, page 23 and Section 4.1.3, page 25.

For examples showing how to compile, link, and run MPI applications, see Chapter 13, page 95 and Chapter 14, page 133.

Note: Programs that use MPI library routines for parallel control and communication should call the `MPI_Finalize()` routine at the conclusion of the program.

For a list of MPI error messages and suggested workarounds, see Appendix D, page 199.

For information about MPI environment variables, see the `intro_mpi`(3) man page.

There are some limitations to Cray XT MPICH2 you should take into consideration:

- There is a name conflict between `stdio.h` and the MPI C++ binding in relation to the names `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`. If your application does not reference these names, you can work around this conflict by using the compiler flag `-DMPICH_IGNORE_CXX_SEEK`. If your application does require these names, as defined by MPI, undefine the names (`#undef SEEK_SET`, for example) prior to the `#include "mpi.h"` statement. Alternatively, if the application requires the `stdio.h` naming, your application should include the `#include "mpi.h"` statement before the `#include <stdio.h>` or `#include <iostream>` statement.
- The following process-creation functions are not supported and, if used, generate aborts at run time:
  - `MPI_Close_port()` and `MPI_Open_port()`
  - `MPI_Comm_accept()`
  - `MPI_Comm_connect()` and `MPI_Comm_disconnect()`
  - `MPI_Comm_spawn()` and `MPI_Comm_spawn_multiple()`
  - `MPI_Comm_get_attr()` with attribute `MPI_UNIVERSE_SIZE`
  - `MPI_Comm_get_parent()`
  - `MPI_Lookup_name()`
  - `MPI_Publish_name()` and `MPI_Unpublish_name()`
- The `MPI_LONG_DOUBLE` data type is not supported.
- The behavior of the MPICH2 function `MPI_Dims_create()` is not consistent with the MPI standard. Therefore, Cray added a special `mpi_dims_create` algorithm to the MPI library. This added function is enabled by default.

3.7 Cray SHMEM Library

The Cray shared memory access (SHMEM) library is a set of logically shared, distributed memory access routines. Cray SHMEM routines are similar to MPI routines; they pass data between cooperating parallel processes. The Cray SHMEM library is implemented on top of the Portals low-level message-passing engine. The Portals interface is transparent to the application programmer.

All Cray XT compilers support SHMEM applications. There are two versions of the SHMEM library available for users of the PGI or PathScale Fortran compilers. One version supports applications where the data size for the Fortran default types integer, real, and logical is 32 bits; the other version supports applications where the size is 64 bits. For further details, see Section 4.1.1.1, page 23 and Section 4.1.3, page 25.

Cray SHMEM routines can be used in programs that perform computations in separate address spaces and that explicitly pass data by means of put and get functions to and from different processing elements in the program. Cray SHMEM routines can be called from Fortran, C, and C++ programs and used either by themselves or with MPI functions.

Portals and the Cray SHMEM library support the following SHMEM atomic memory operations:

- atomic swap
- atomic conditional swap
- atomic fetch and increment
- atomic fetch and add
- atomic lock

An operation is atomic if the steps cannot be interrupted and are done as a unit.

When running on Catamount, you can use the `yod` command line options `-stack`, `-heap`, and `-shmem` to control the size (in bytes) of the stack, private heap, and symmetric heap, respectively. See the `yod`(1) man page for details. On Catamount, SHMEM applications can use all available memory per node (total memory minus memory for the kernel and the process control thread (PCT)). SHMEM does not impose any restrictions on stack, heap, or symmetric heap memory regions.

When running on CNL, the environment variable XT_LINUX_SHMEM_HEAP_SIZE can be used to control the size (in bytes) of the private heap. The size of the stack is limited by the value of stacksize in a process' limits, if this is not unlimited. If this limit is set to unlimited, then the default size of the stack is 16 MB, unless the user sets the environment variable XT_LINUX_SHMEM_STACK_SIZE, which specifies the desired size of the stack in bytes.

The environment variable XT_SYMMETRIC_HEAP_SIZE can be used when running on either Catamount or CNL to control the size (in bytes) of the symmetric heap.

Note: To build, compile, and run Cray SHMEM applications, you need to call start_pes(int *npes*) or shmem_init() as the first Cray SHMEM call and shmem_finalize() as the last Cray SHMEM call.

For examples showing how to compile, link, and run SHMEM applications, see Chapter 13, page 95 and Chapter 14, page 133.

When using SHMEM functions, you should be aware of the following performance issues:

- The performance of strided operations is poor. The Portals network protocol stack on Cray XT series is optimized for block transfers. It does not support efficient access of non-contiguous remote memory. Repackaging data into contiguous blocks in the application and then calling a shmem_put() or shmem_get() function will lead to better performance than calling strided operations. You may want to try this option if your application uses strided SHMEM operations.
- The performance of atomic operations is poor because Cray XT series systems do not provide hardware support for atomic memory operations. Atomic memory operations should not be used for high fan-in synchronization because the injection rate is much larger than the processing rate, leading to a buildup of requests and, in turn, degraded performance.
- Cray XT series systems do not support barrier operations in hardware or firmware. The barrier functions are implemented in software and are relatively slow. Cray recommends that you minimize the use of barriers.
- Avoid the following type of constructs:

```
while (remval != 0) {
 shmem_get64(&remval, &rem_flag, 1, pe);
 }
```

They can severely tax the Portals network protocol stack, particularly if many processes are spinning on a variable at a single target process. If possible, use other synchronization mechanisms that rely on spinning on local memory.

3.8 OpenMP Library

The Cray XT Series system supports version 2.5 of the OpenMP Application Program Interface standard. OpenMP is a shared-memory parallel programming model that application developers can use to create and distribute work using threads. In addition to library routines, OpenMP provides Fortran directives, C and C++ pragmas, and environment variables. The PGI, PathScale, and GNU compilers support OpenMP.

To use OpenMP, you need to include the appropriate OpenMP option on the compiler command line. The compiler command options are:

PGI	`-mp=nonuma`
PathScale	`-mp`
GCC	`-fopenmp`

You also need to set the `OMP_NUM_THREADS` environment variable to the number of threads in the team.

The number of processors hosting OpenMP threads at any given time is fixed at program startup and specified by the `aprun -d` *depth* option (see Section 7.1, page 53 for further information).

For an example showing how to compile, link, and run OpenMP applications, see Example 10, page 106.

OpenMP applications can be used in hybrid OpenMP / MPI applications but may not cross node boundaries. In OpenMP / MPI applications, MPI calls can be made from master or sequential regions but not parallel regions. OpenMP is supported on CNL but not Catamount.

For further information about launching OpenMP applications, see the `aprun`(1) man page. For further information about OpenMP functions, see the OpenMP website (`http://www.openmp.org`), the PGI website (`http://www.pgroup.com/`), the PathScale website (`http://www.pathscale.com/`), or the GNU OpenMP website (`http://gcc.gnu.org/projects/gomp/`).

Programming Considerations [4]

The manuals and man pages for third-party and open source Cray XT series Programming Environment products provide platform-independent descriptions of product features. This chapter provides information specific to Cray XT series systems that you should consider when using those products to develop CNL or Catamount applications. The following sections describe general programming considerations, Catamount-specific programming considerations, and CNL-specific programming considerations.

4.1 General Programming Considerations

This section describes product features that apply to all applications.

4.1.1 PGI Compilers

When using the PGI compilers, you should be aware of the following factors.

4.1.1.1 Default MPICH2 and SHMEM Libraries

Users of the PGI Fortran compiler have the option of promoting default integer, real, and logical operations to 64-bit precision. By including the `-default64` option on the `ftn` command line, you pass the `-i8` and `-r8` options to the compiler. The `-i8` option directs the compiler to use 64 bits for the data size of default integer and logical operations. The `-r8` option directs the compiler to use 64 bits for the data size of default real variables.

All Fortran source files for the application containing default integer, logical, real, or complex variables must be compiled this way. In addition, for MPI applications the `-default64` option directs the linker to use the default64 version of the MPI library. For SHMEM applications, the `-default64` option directs the linker to use the default64 version of the SHMEM library.

Remember to link in default64 mode. If you compile using `-default64` but omit the `-default64` option when linking the compiled object files into an executable, the compiler will attempt to link to the default32 libraries, and the resulting executable probably will not run.

Note: The sizes of data types that use explicit kind and star values are not affected by this option.

For further information, see the ftn(1) man page.

4.1.1.2 Unsupported C++ Header Files

PGI does not provide a complete set of the old C++ Standard Library and STL header files. PGI C++ does support some old header files (iostream.h, exception.h, iomanip.h, ios.h, istream.h, ostream.h, new.h, streambuf.h, strstream.h, and typeinfo.h), which include their C++ Standard Library counterpart.

To use an unsupported header file, you can:

- Delete the .h. For example, change <vector.h> to <vector>, **or**
- Create your own *headerfile*.h file and use the -I compiler option to direct the compiler to access the header file in your directory:

```
#ifndef __VECTOR_H
#define __VECTOR_H
#include <vector>
using std::vector;
#endif
```

4.1.1.3 Restrictions on Large Data Objects

The PGI compilers support data objects larger than 2 GB. However, the Cray XT series Programming Environment has restrictions in this area because the user-level libraries (MPI, SHMEM, and LibSci) are compiled in the small memory model.

The only way to build an application with data objects larger than 2 GB is to limit the static data sections to less than 2 GB by converting static data to dynamically allocated data.

4.1.1.4 The FORTRAN STOP Message

For PGI Fortran, the stop statement writes a FORTRAN STOP message to standard output. In a parallel application, the FORTRAN STOP message is written by every process that executes the stop statement: potentially, every process in the communicator space. This is not scalable and will cause performance and, potentially, reliability problems in applications of very large scale.

You can turn off the STOP message by using the NO_STOP_MESSAGE environment variable. For examples, see Example 9, page 105 and Example 27, page 142.

4.1.1.5 Unsupported Compiler Command Options

The following PGI compiler command options are not supported on Cray XT series systems:

- `-mprof=mpi`
- `-Mmpi`
- `-Mscalapack`

4.1.1.6 Suppressing Vectorization

Cray XT series systems support the following methods of suppressing vectorization in PGI applications:

- The `-Mnovect` compiler option suppresses vectorization for the entire source file.
- The `!pgi$r novector` directive or `#pragma routine novector` statement placed before the start of a routine suppresses vectorization for the entire routine.
- The `!pgi$ novector` directive or `#pragma loop novector` statement placed before a loop suppresses vectorization for the loop. This directive does not suppress vectorization for loops nested inside the targeted loop, so in most cases you should apply the directive to innermost loops.

For further information, see the *PGI User's Guide*.

4.1.2 PGI Debugger

The PGI debugger, PGDBG, is not supported on Cray XT series systems.

4.1.3 PathScale Fortran Compiler

Users of the PathScale Fortran compiler have the option of promoting default integer, real, and logical operations to 64-bit precision. By including the `-default64` option on the `ftn` command line, you pass the `-i8` and `-r8` options to the compiler. The `-i8` option directs the compiler to use 64 bits for the data size of default integer and logical operations. The `-r8` option directs the compiler to use 64 bits for the data size of default real variables.

All Fortran source files for the application containing default integer, logical, real, or complex variables must be compiled this way. In addition, for MPI applications the `-default64` option directs the linker to use the default64 version of the MPI library. For SHMEM applications, the `-default64` option directs the linker to use the default64 version of the SHMEM library.

Remember to link in default64 mode. If you compile using the `-default64` option but omit the `-default64` option when linking the compiled object files into an executable, the compiler will attempt to link to the default32 libraries, and the resulting executable probably will not run.

Note: The sizes of data types that use explicit kind and star values are not affected by this option.

For further information, see the `ftn`(1) man page.

4.1.4 Little-endian Support

The Cray XT series system supports little-endian byte ordering. The least significant value in a sequence of bytes is stored first in memory.

4.1.5 Portals Message Size Limit

A single Portals message cannot be longer than 2 GB.

4.1.6 Shared Libraries

The Cray XT series systems currently do not support dynamic loading of executable code or shared libraries. Also, the related `LD_PRELOAD` environment variable is not supported.

4.2 CNL Programming Considerations

This section describes the factors you need to take into consideration when developing applications to be run on CNL compute nodes.

4.2.1 CNL glibc Functions

CNL provides limited support of the process control functions such as `popen()`, `fork()`, and `exec()`; the resulting processes execute in the limited RAM disk environment on each compute node.

The exec() function can execute the scp and ksh commands and the following BusyBox commands:

ash	gunzip	nice
cat	kill	ping
chmod	killall	ps
chown	ln	renice
cp	rm	cpio
ls	tail	dmesg
mkdir	test	free
vi	grep	more
zcat		

For further information, see the busybox(1) man page.

CNL supports the cpuinfo and meminfo /proc files. These files contain information about your compute node.

CNL glibc does not support:

- The getgrgid(), getgrnam(), getpwnam(), and getpwuid() functions.
- Customer-provided functions that require a daemon.

Appendix A, page 181 lists the glibc functions that CNL supports. The glibc functions that CNL does not support are so noted in their man pages.

4.2.2 I/O Support

The I/O operations allowed in CNL applications are Fortran, C, and C++ I/O calls; Cray MPICH2, Cray SHMEM, and OpenMP I/O functions; and the underlying Linux Lustre client I/O functions.

In Catamount, I/O is possible to any file system accessible to yod. Lustre I/O is handled as a special case. In CNL, only I/O to Lustre is supported. Files in other remote file systems cannot be accessed. One exception is the handling of stdin, stdout, and stderr.

The aprun utility handles stdin, stdout, and stderr. The aprun file descriptor 0 forwards stdin data to processing element 0 (PE 0) only; stdin is closed on all other PEs. The stdout and stderr data from all PEs is sent to aprun, which forwards the data to file descriptors 1 and 2.

Files local to the compute node, such as ones in /proc or /tmp, can be accessed by a CNL application.

4.2.3 External Connectivity

Cray XT series systems support external connectivity to or from compute nodes running CNL. You can use IP functions in your programs to access network services. To determine if your site has configured CNL compute nodes for network connectivity, see your system administrator.

4.2.4 Timing Functions

CNL supports the following timing functions:

- CPU timers. CNL supports the Fortran cpu_time() function. The Fortran cpu_time(*time*) intrinsic subroutine returns the processor time, where *time* has a data type of real4 or real8. The magnitude of the value returned by cpu_time() is not necessarily meaningful. You call cpu_time() before and after a section of code; the difference between the two times is the amount of CPU time (in seconds) used by the program.
- Elapsed time counter. CNL supports the MPI_Wtime() and MPI_Wtick()functions and the Fortran system_clock() intrinsic subroutine.

  The MPI_Wtime() function returns the elapsed time. The MPI_Wtick() function returns the resolution of MPI_Wtime() in seconds.

  CNL does not support the dclock() or etime() functions.

4.2.5 Signal Support

The aprun utility catches and forwards the SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGABRT, SIGUSR1, and SIGUSR2 signals to an application. For further information, see Section 7.8, page 58.

4.2.6 Core Files

When an application fails on CNL, one core file is generated for the first failing process. An application generates no core file at all if a file named `core` already exists in the current directory.

4.2.7 Page Size

CNL supports a single page size of 4 KB.

4.2.8 Resource Limits

Memory limits are defined by the node default or the `aprun -m` option. Time limits are inherited from the `aprun` process limits or specified with the `aprun -t` option. Other limits are inherited from the limits of `aprun`. All limits apply to individual processing elements; there are no aggregate application limits that can be specified with `aprun` options.

4.2.9 One Application Per Node Limitation

The Cray XT series currently does not support running more than one CNL application on a dual-core compute node.

4.2.10 Parallel Programming Models

The MPI, SHMEM, and OpenMP parallel programming models are supported on CNL applications.

4.2.11 Modified Copy-on-write Process

Under Linux, `fork()` uses a copy-on-write process to conserve time and memory resources. When a process forks a child process, most of the pages in the parent process' address space are initially shared with the child process. The parent and child processes can continue sharing a page until one of the processes tries to modify the page. At that point, the process modifying the page creates a new page for its private use, copies the previously-shared page's data into it, and continues to use this new page instead of the previously-shared page. The previously-shared page now belongs solely to the other process.

The copy-on-write process can adversely affect Cray XT user applications that use Portals. To correct this problem, Cray modified the Portals kernel to perform a partial copy when a process forks a child process. For each region of a process' address space that is registered with Portals for Remote Direct Memory Access (RDMA), the first and last page of the region are copied to a private page in the child's address space as the fork occurs. This ensures that Portals can continue to transfer data using these pages in the parent's address space, and also ensures that any data residing on these pages that were not intended for Portals transfers (such as heap variables) can be referenced in the child's address space.

The implications for application developers are:

- Pages in the middle of a Portals memory region (likely maps to any large MPI message buffers) are not accessible in the child process. You should copy the necessary data out of the parent's message buffer before forking.
- More memory is allocated and copied than in a normal fork. This could cause unexpected memory exhaustion if you have many Portals memory regions.

4.3 Catamount Programming Considerations

This section describes the factors you need to take into consideration when developing applications to be run on Catamount compute nodes.

4.3.1 Catamount glibc Functions

Because Catamount is designed specifically to provide critical support to high-speed computational applications, its functionality is limited in certain areas where the service nodes are expected to take over. In particular, glibc on Catamount does not support:

- Dynamic process control (such as `exec()`, `popen()`, `fork()`, or `system` library calls).
- Threading.
- The `/proc` files such as `cpuinfo` and `meminfo`. (These files contain information about your login node.)
- The `ptrace()` system call.

- The `mmap()` function. If `mmap()` is called, a skeleton function returns `-1`. You should use `malloc()` instead of `mmap()` if the `mmap()` call is using the `MAP_ANONYMOUS` flag; `malloc()` is not an appropriate replacement for `mmap()` calls that use the `MAP_FIXED` or `MAP_FILE` flag. If you do use `malloc()`, be aware that you may have to resolve data alignment issues. See the `malloc()` man page for details.

  Note: The Cray XT series system provides two implementations of `malloc()`: Catamount `malloc()` and GNU `malloc()`. Catamount provides a custom implementation of the `malloc()` function. This implementation is tuned to Catamount's non-virtual-memory operating system and favors applications allocating large, contiguous data arrays. The function uses a first-fit, last-in-first-out (LIFO) linked list algorithm. For information about gathering statistics on memory usage, see the `heap_info`(3) man page. In some cases, GNU `malloc()` may improve performance.

- The `profil()` function.
- Any of the `getpwd*()`, `getgr*()`, and `getpw*()` families of library calls.
- Terminal control.
- Customer-provided functions that require a daemon.
- Any functions that require a database, such as Network Block Device (NDB) functions. For example, there is no support for the `uid` and `gid` family of queries that are based on the NDB functions.
- There is limited support for signals and `ioctl()`. See the man page for details.

Appendix B, page 187 lists the glibc functions that Catamount supports. The glibc functions that Catamount does not support are so noted in their man pages.

4.3.2 I/O Support

I/O support for Catamount applications is limited. The only operations allowed are Fortran, C, and C++ I/O calls; Cray MPICH2 and Cray SHMEM I/O functions; and the underlying Catamount (libsysio) and Lustre (liblustre) I/O functions.

Application programmers should keep in mind the following behaviors:

- I/O is offloaded to the service I/O nodes. The `yod` application launcher handles `stdin`, `stderr`, and `stdout`. For more information, see Section 8.6, page 64.
- Calling an I/O function such as `open()` with a bad address causes the application to fail with a page fault. On the service nodes, a bad address causes the function to set `errno = EFAULT` and return `-1`.
- Catamount does not support I/O on named pipes.

The following sections describe techniques you can use to improve I/O performance.

4.3.2.1 Improving Fortran I/O Performance

To increase buffer size in a Fortran program, use the `setvbuf3f()` function:

```
integer function setvbuf3f(lu, type, size)
```

Table 2. `setvbuf3f()` Arguments

Argument	Description
`integer` *`lu`*	The logical unit
`integer` *`type`*	`0` — Full buffering
	`1` — Line buffering
	`2` — No buffering
`integer` *`size`*	The size of the new buffer

The `setvbuf3f()` function returns 0 on success, nonzero on failure. For further information, see the `setbuf`(3) man page.

4.3.2.2 Improving C++ I/O Performance

The standard stream I/O facilities defined in the Standard C++ header file `<iostream>` are unbuffered. You can use the routine `pubsetbuf()` to specify a buffer for I/O. Example 29, page 144 shows how `pubsetbuf()` can improve performance.

I/O-to-file streams defined in <fstream> are buffered with a default buffer size of 4096. You can use pubsetbuf() to specify a buffer that has a different size. You must specify the buffer size before the program performs a read or write to the file; otherwise, the call to pubsetbuf() is ignored and the default buffer is used. Example 30, page 145 shows how to use pubsetbuf() to specify a buffer for <fstream> file I/O. Avoid calls to member function endl to prevent the buffer from being flushed.

4.3.2.3 Improving stdio Performance

By default, stdin, stdout, and stderr are unbuffered. Under Catamount, this limits the data transfer rate to approximately 10 bytes per second because read and write calls are offloaded to yod. To improve performance, call setvbuf() to buffer stdin input or stdout/stderr output. For an example showing how to improve stdio performance, see Example 31, page 147.

4.3.2.4 Improving Large File, Sequential I/O Performance

IOBUF is an I/O buffering library that can reduce the I/O wait time for programs that read or write large files sequentially. IOBUF intercepts standard I/O calls such as fread() and fopen() and replaces the stdio layer of buffering with a replacement layer of buffering, thus improving program performance by enabling asynchronous prefetching and caching of file data. In addition, IOBUF can gather run time statistics and print a summary report of I/O activity for each file.

No program source changes are needed to use IOBUF. Instead, you relink your program with the IOBUF library and set one or more environment variables.

To use IOBUF, follow these steps:

1. Load the iobuf module:

   ```
   % module load iobuf
   ```

2. Relink the program.

3. Set the IOBUF_PARAMS environment variable.

   The IOBUF_PARAMS environment variable specifies patterns for selecting I/O files and sets parameters for buffering. If this environment variable is not set, the default state is no buffering and the I/O call is passed on to the next layer without intervention.

The general format of the IOBUF_PARAMS environment variable is a comma-separated list of specifications:

```
IOBUF_PARAMS 'spec1,spec2,spec3,...'
```

Each specification begins with a file name pattern. When a file is opened, the list of specifications is scanned and the first matching file name pattern is selected. If no pattern matches, the file is not buffered. The file name pattern follows standard shell pattern matching rules. For example, to buffer stdout, use:

```
% setenv IOBUF_PARAMS '%stdout'
```

4. Execute the program.

Note: IOBUF works with PGI Fortran programs but does not work with PathScale Fortran or GNU Fortran programs. Also, IOBUF works with the PGI, PathScale, and GNU C compilers. IOBUF works with C++ programs that use stdio but does not work with the C++ standard buffered I/O stream class <iostream>.

C programs that use POSIX-style I/O calls like open(), read(), write(), and close() are not affected by IOBUF. A workaround is to replace POSIX I/O calls in the C program with their equivalent IOBUF-specific calls. The IOBUF calls are identical to their POSIX counterparts but are prefixed with iobuf_.

For further information, see the iobuf(3) man page.

4.3.2.5 Using Stride I/O Functions to Improve Performance

You can improve file I/O performance of C and C++ programs by using the readx(), writex(), ireadx(), and iwritex() stride I/O functions. For further information, see the man pages.

4.3.2.6 Reducing Memory Fragmentation

In past releases, small memory allocations could become interspersed throughout memory, preventing the allocation of very large arrays (that is, arrays larger than half of available memory). To solve this problem, small allocations (those less than or equal to 100 MB, by default) are still allocated into the beginning of the first available free area of memory, but large allocations are now allocated into the end of the last available free area. This allows very large arrays to be allocated/freed in a separate area of memory, making memory fragmentation less likely.

You can use the CATMALLOC_LARGE_ALLOC_SIZE environment variable to change the default small versus large delineation line.

4.3.3 External Connectivity

Cray XT does not support external connectivity to or from compute nodes running Catamount. Pipes, sockets, remote procedure calls, or other types of TCP/IP communication are not supported. The Cray MPICH2, Cray SHMEM, and OpenMP parallel programming models and the underlying Portals interface are the only supported communication mechanisms.

4.3.4 Timing Functions

Catamount supports the following timing functions:

- Interval timer. Catamount supports the `setitimer ITIMER_REAL` function. It does not support the `settimer ITIMER_VIRTUAL` or the `setitimer ITIMER_PROF` function. Also, Catamount does not support the `getitimer()` function.
- CPU timers. Catamount supports the glibc `getrusage()` and the Fortran `cpu_time()` functions. For C and C++ programs, `getrusage()` returns the current resource usages of either `RUSAGE_SELF` or `RUSAGE_CHILDREN`. The Fortran `cpu_time(`*`time`*`)` intrinsic subroutine returns the processor time, where *time* has a data type of `real4` or `real8`. The magnitude of the value returned by `cpu_time()` is not necessarily meaningful. You call `cpu_time()` before and after a section of code; the difference between the two times is the amount of CPU time (in seconds) used by the program.
- Elapsed time counter. The `dclock()`, Catamount `clock()`, and `MPI_Wtime()` functions and the `system_clock()` Fortran intrinsic subroutine calculate elapsed time. The `etime()` function is not supported.

  The `dclock()` value rolls over approximately every 14 years and has a nominal resolution 100 nanoseconds on each node.

  Note: The `dclock()` function is based on the configured processor frequency, which may vary slightly from the actual frequency. The clock frequency is not calibrated. Furthermore, the difference between configured and actual frequency may vary slightly from processor to processor. Because of these two factors, accuracy of the `dclock()` function may be off by as much as +/-50 microseconds/second or 4 seconds/day.

  The `system_clock()` function has a resolution of 1000 ticks per second.

The `clock()` function is now supported on Catamount; it estimates elapsed time as defined for `dclock()`. The Catamount `clock()` function is not the same as the Linux `clock()` function. The Linux `clock()` function measures processor time used. For Catamount compute node applications, Cray recommends that you use the `dclock()` function or an intrinsic timing routine in Fortran such as `cpu_time()` instead of `clock()`. For further information, see the `dclock`(3) and `clock`(3) man pages.

The `MPI_Wtime()` function returns the elapsed time. The `MPI_Wtick()` function returns the resolution of `MPI_Wtime()` in seconds. For an example showing how to use `dclock()` to calculate elapsed time, see Example 28, page 143.

4.3.5 Signal Support

In previous Cray XT series releases, Catamount did not correctly provide extra arguments to signal handlers when the user requested them through `sigaction()`. Signal handlers installed through `sigaction()` have the prototype:

```
void (*handler) (int, siginfo_t *, void *)
```

which allows a signal handler to optionally request two extra parameters. On Catamount compute nodes, these extra parameters are provided in a limited fashion when requested.

The `siginfo_t` pointer points to a valid structure of the correct size but contains no data.

The `void *` parameter points to a `ucontext_t` structure. The `uc_mcontext` field within that structure is a platform-specific data structure that, on compute nodes, is defined as a `sigcontext_t` structure. Within that structure, the general purpose and floating-point registers are provided to the user. You should rely on no other data.

For a description of how `yod` propagates signals to running applications, see Section 8.7, page 64.

4.3.6 Core Files

By default, when an application fails on Catamount, only one core file is generated: that of the first failing process. For information about overriding the defaults, see the `core`(5) man page. Use caution with the overrides because dumping core files from all processes is not scalable.

4.3.7 Page Size

The `yod -small_pages` option allows you to specify 4 KB pages instead of the default 2 MB pages. Locality of reference affects the optimum choice between the default 2 MB pages and the 4 KB pages. Because it is often difficult to determine how the compiler is allocating your data, the best approach is to try both the default and the `-small_pages` option and compare performance numbers.

Note: For each 1 GB of memory, 2 MB of page table space are required.

The Catamount `getpagesize()` function returns `4 KB`.

4.3.8 Resource Limits

Because a Catamount application has dedicated use of the processor and physical memory on a compute node, many resource limits return `RLIM_INFINITY`. Keep in mind that while Catamount itself has no limitation on file size or the number of open files, the specific file systems on the Linux service partition may have limits that are unknown to Catamount.

On Catamount, the `setrlimit()` function always returns `success` when given a valid resource name and a non-NULL pointer to an `rlimit` structure. The `rlimit` value is never used because Catamount gives the application dedicated use of the processor and physical memory.

4.3.9 Parallel Programming Models

The MPI and SHMEM parallel programming models are supported on Catamount applications. OpenMP is not supported on Catamount.

Compiler Overview [5]

The Cray XT series Programming Environment includes Fortran, C, and C++ compilers from PGI, GNU, and PathScale. You access the compilers through Cray XT series compiler drivers. The compiler drivers perform the necessary initializations and load operations, such as linking in the header files and system libraries (`libc.a` and `libmpich.a`, for example) before invoking the compilers.

5.1 Setting Your Target Architecture

Before you begin to compile programs, you must verify that the target architecture is set correctly. The target architecture is used by the compilers and linker in creating executables to run on either CNL or Catamount compute nodes; it is set automatically when you log in. If the compute nodes are running CNL, the `xtpe-target-cnl` module is loaded and the `XTPE_COMPILE_TARGET` environment variable is set to `linux`. If the compute nodes are running Catamount, the `xtpe-target-catamount` module is loaded and `XTPE_COMPILE_TARGET` is set to `catamount`.

To determine the current target architecture, use the `module list` command. Either `xtpe-target-cnl` or `xtpe-target-catamount` will be loaded.

You cannot run a CNL application on compute nodes running Catamount nor a Catamount application on compute nodes running CNL. However, you can create CNL or Catamount executables at any time by configuring your environment properly.

For example, if the target architecture is `catamount` and you want to create executable to run under CNL, swap `xtpe-target` modules:

```
% module swap xtpe-target-catamount xtpe-target-cnl
```

5.2 Using Compilers

The syntax for invoking the compiler drivers is:

```
% compiler_command [PGI_options|GCC_options|PathScale_options]
filename,...
```

For example, to use the PGI Fortran compiler to compile `prog1.f90` and create default executable `a.out` to be run on CNL compute nodes, first verify that the following modules have been loaded:

```
PrgEnv-pgi
xtpe-target-cnl
```

Then use the following command:

```
% ftn prog1.f90
```

If you next want to use the PathScale C compiler to compile `prog2.c` and create default executable `a.out` to be run on Catamount compute nodes, use the following commands:

```
% module swap PrgEnv-pgi PrgEnv-pathscale
% module swap xtpe-target-cnl xtpe-target-catamount
```

Then invoke the C compiler:

```
% cc prog2.c
```

Note: Verify that your CNL and Catamount executables are stored in separate directories or differentiated by file name. If you try to run a CNL application when Catamount is running or a Catamount application when CNL is running, your application will abort.

5.2.1 Using PGI Compilers

To use the PGI compilers, run the `module list` command to verify that the `PrgEnv-pgi` module is loaded. If it is not, use a `module swap` command, such as:

```
% module swap PrgEnv-gnu PrgEnv-pgi
```

`PrgEnv-pgi` loads the product modules that define the system paths and environment variables needed to use the PGI compilers.

For a description of new and modified PGI compiler features, see the *PGI Server 7.0 and Workstation 7.0 Installation and Release Notes*.

Note: When linking in ACML routines, you must compile and link all program units with `-Mcache_align` or an aggregate option such as `fastsse`, which incorporates `-Mcache_align`.

The commands for invoking the PGI compilers and the source file extensions are:

Table 3. PGI Compiler Commands

Compiler	Command	Source File
C compiler	`cc`	*filename*`.c`
C++ compiler	`CC`	*filename*`.C`
Fortran 90/95 compiler	`ftn`	*filename*`.f` (fixed source) *filename*`.f90`, *filename*`.f95`, *filename*`.F95` (free source)
FORTRAN 77 compiler	`f77`	*filename*`.f77`

Caution: To invoke a PGI compiler, use the `cc`, `CC`, `ftn`, or `f77` command. If you invoke a compiler directly using a `pgcc`, `pgCC`, `pgf95`, or `pgf77` command, the resulting executable will not run on a Cray XT series system.

The `cc`(1), `CC`(1), `ftn`(1), and `f77`(1) man pages contain information about the compiler driver commands, whereas the `pgcc`(1), `pgCC`(1), `pgf95`(1), and `pgf77`(1) man pages contain descriptions of the PGI compiler command options.

The *PGI User's Guide* and the *PGI Fortran Reference* manual include information about compiler features unique to Cray (see `http://www.pgroup.com/resources/docs.htm`).

Examples of compiler commands:

```
% cc -c myCprog.c
% CC -o my_app myprog1.o myCCprog.C
% ftn -fastsse -Mipa=fast prog.f sample1.f
% cc -c c1.c
% ftn -o app1 f1.f90 c1.o
```

To verify that you are using the correct version of a compiler, use the `-V` option on a `cc`, `CC`, `ftn`, or `f77` command.

Note: The `-Mconcur` (auto-concurrentization of loops) option documented in the PGI manuals is not supported on Cray XT series systems.

5.2.2 Using GNU Compilers

To use the GNU compilers, run the `module list` command to verify that the `PrgEnv-gnu` module is loaded. If it is not, use a `module swap` command, such as:

```
% module swap PrgEnv-pgi PrgEnv-gnu
```

`PrgEnv-gnu` loads the product modules that define the system paths and environment variables needed to use the GNU compilers.

Both GCC 3.3.3 and 4.2.1 are supported. GCC 3.3.3 includes the FORTRAN 77, C, and C++ compilers; GCC 4.2.1 includes the Fortran 95, C, and C++ compilers. The `f77` command compiles FORTRAN 77 programs. You can use the `ftn` command to compile either Fortran 95 or FORTRAN 77 programs.

To determine whether the desired GCC module is loaded, use the `module list` command. If the desired module is not loaded, use the `module swap` command, such as:

```
% module swap gcc/3.3.3 gcc/4.2.1
```

The commands for invoking the GNU compilers and the source file extensions are:

Table 4. GNU Compiler Commands

Compiler	Command	Source File
C compiler	`cc`	*filename*`.c`
C++ compiler	`CC`	*filename*`.cc`, *filename*`.c++`, *filename*`.C`
Fortran 95 and FORTRAN 77 compilers (GCC 4.1.1 and later)	`ftn`	*filename*`.f`, *filename*`.f90`, *filename*`.f95`
FORTRAN 77 compiler (GCC 3.2.3 only)	`f77`	*filename*`.f`

The *Using the GNU Compiler Collection (GCC)* manual provides general information about the GNU compilers. The *GNU Fortran 95 Compiler Manual* and the *G77 Manual* include information about compiler features unique to Cray (see `http://gcc.gnu.org/onlinedocs/`).

Caution: To invoke a GNU compiler, use the `cc`, `CC`, `ftn`, or `f77` command. If you invoke a compiler directly using a `gcc`, `g++`, `gfortran`, or `g77` command, the resulting executable will not run on a Cray XT series system.

The `cc`(1), `CC`(1), `ftn`(1), and `f77`(1) man pages contain information about the compiler driver commands, whereas the `gcc`(1), `g++`(1), `gfortran`(1), and `g77`(1) man pages contain descriptions of the GNU C compiler command options.

Examples of GNU compiler commands (assuming the `PrgEnv-gnu` module is loaded):

```
% cc -c c1.c
% CC -o app1 prog1.o C1.C
% ftn -o mpiapp mpi1.f mpi2.o
% f77 -o sample1 sample1.f
```

To verify that you are using the correct version of a GNU compiler, use the `--version` option on a `cc`, `CC`, `ftn`, or `f77` command.

Note: To use CrayPat with a GNU program to trace functions, use the `-finstrument-functions` option instead of `-Mprof=func` when compiling your program.

5.2.3 Using PathScale Compilers

To use the PathScale compilers, run the `module list` command to verify that the `PrgEnv-pathscale` module is loaded. If it is not, use a `module swap` command, such as:

```
% module swap PrgEnv-pgi PrgEnv-pathscale
```

`PrgEnv-pathscale` loads the product modules that define the system paths and environment variables needed to use the PathScale compilers.

The commands for invoking the PathScale compilers and the source file extensions are:

Table 5. PathScale Compiler Commands

Compiler	Command	Source File
C compiler	`cc`	*filename*`.c`
C++ compiler	`CC`	*filename*`.CC`
		filename`.cc`
		filename`.cpp`
		filename`.cxx`
Fortran 90/95 and FORTRAN 77 compilers	`ftn`	*filename*`.f` (fixed source, no preprocessing)
		filename`.f90` (free source, no preprocessing)
		filename`.f95` (free source, no preprocessing)
		filename`.F` (fixed source, preprocessing)
		filename`.F90` (free source, preprocessing)
		filename`.F95` (free source, preprocessing)

To verify that you are using the correct version of a PathScale compiler, use the `-version` option on a `cc` , `CC`, or `ftn` command.

Caution: To invoke a PathScale compiler, use either the `cc`, `CC`, or `ftn` command. If you invoke a compiler directly using a `pathcc`, `pathCC`, or `path95` command, the resulting executable will not run on a Cray XT series system.

The `cc`(1), `CC`(1), and `ftn`(1) man pages contain information about the compiler driver commands, whereas the `pathcc`(1), `pathCC`(1), and `path95`(1) man pages contain descriptions of the PathScale compiler command options.

The `eko`(7) man page gives the complete list of options and flags for the PathScale compiler suite.

Examples of PathScale compiler commands (assuming the `PrgEnv-pathscale` module is loaded):

```
% cc -c c1.c
% CC -o app1 prog1.o C2.C
% ftn -o sample1 sample1.f
```

For more information about using the compiler commands, see the PathScale manuals at `http://www.pathscale.com/docs/html` and the following man pages:

- Introduction to PathScale compilers: `pathscale-intro`(1) man page
- C compiler: Cray `cc`(1) man page and PathScale `pathcc`(1) and `eko`(7) man pages
- C++ compiler: Cray `CC`(1) man page and PathScale `pathCC`(1) and `eko`(7) man pages
- Fortran compiler: Cray `ftn`(1) man page and PathScale `path95`(1) and `eko`(7) man pages

Getting Compute Node Status [6]

Before running applications, you should check the status of the compute nodes.

First, use either the `xtprocadmin -A` or `cnselect -L osclass` command to find out whether CNL or Catamount is running on the compute nodes.

For the `xtprocadmin -A` report, the `OS` field value is `CNL` or `Catamount` for all compute nodes, and `service` for all service nodes. For the `cnselect -L osclass` report, `osclass` is `1` for Catamount and `2` for CNL.

```
% xtprocadmin -A
   NID    (HEX)    NODENAME     TYPE ARCH        OS CORES AVAILMEM   PAGESZ CLOCKMHZ
<snip>
    93     0x5d  c0-0c2s7n1  compute   xt       CNL     1     2000     4096     2400

    94     0x5e  c0-0c2s7n2  compute   xt       CNL     1     2000     4096     2400

    95     0x5f  c0-0c2s7n3  compute   xt       CNL     1     2000     4096     2400

   128     0x80  c1-0c0s0n0  service   xt (service)     1     4000     4096     2400

   131     0x83  c1-0c0s0n3  service   xt (service)     1     4000     4096     2400

   132     0x84  c1-0c0s1n0  service   xt (service)     1     2000     4096     2400
<snip>

% cnselect -L osclass
2
```

Then use the `xtshowmesh` or `xtshowcabs` command. These utilities show node status (up or down, allocated to interactive or batch processing, free or in use). Each character in the display represents a single node. For systems running a large number of jobs, more than one character may be used to designate a job.

```
% xtshowmesh
Compute Processor Allocation Status as of Wed Sep 12 08:06:28 2007

        C 0 (X dir) C 1 (X dir) C 2 (X dir) C 3 (X dir) C 4 (X dir) C 5 (X dir)
```

```
Z dir-> 01234567   01234567   01234567   01234567   01234567   01234567
Y dir 0  SSSSS--   --------
      1       ac   --------
      2       b-   --------
      3 SSSSSS--   --------
      4 --------   --------
      5 --------   --------
      6 d-------   --------
      7 --------   --------
      8 --------   --------
      9 --------   --------
     10 --------   --------
     11 --------   --------

      C 2 (X dir) C 3 (X dir) C 4 (X dir) C 5 (X dir) C 6 (X dir) C 7 (X dir)

Z dir-> 01234567   01234567   01234567   01234567   01234567   01234567
Y dir 0 --------   --------
      1 --------   --------
      2 --------   --------
      3 --------   --------
      4 --------   --------
      5 --------   --------
      6 --------   --------
      7 --------   --------
      8 --------   --------
      9 --------   --------
     10 --------   --------
     11 --------   --------

      C 4 (X dir) C 5 (X dir) C 6 (X dir) C 7 (X dir) C 8 (X dir) C 9 (X dir)
Z dir-> 01234567   01234567   01234567   01234567   01234567   01234567
Y dir 0
      1
      2
      3
      4
      5
      6
      7
      8
      9
```

```
      10
      11                   S

       C 6 (X dir) C 7 (X dir) C 8 (X dir) C 9 (X dir)

  Z dir-> 01234567   01234567   01234567   01234567
  Y dir 0 --------   --------
        1 --------   --------
        2 --------   --------
        3 --------   --------
        4 --------   --------
        5 --------   --------
        6 --------   --------
        7 --------   --------
        8 --------   --------
        9 --------   --------
       10 --------   --------
       11 --------   --------

       C 8 (X dir) C 9 (X dir)

  Z dir-> 01234567   01234567
  Y dir 0 --------   --------
        1 --------   --------
        2 --------   --------
        3 --------   --------
        4 --------   --------
        5 --------   --------
        6 --------   --------
        7 --------   --------
        8 --------   --------
        9 --------   --------
       10 --------   --------
       11 --------   --------

Legend:
   nonexistent node                S  service node
;  free interactive compute CNL    -  free batch compute node CNL
A  allocated, but idle compute node ?  suspect compute node
X  down compute node               Y  down or admindown service node
Z  admindown compute node          R  node is routing

Available compute nodes:       0 interactive,   740 batch
```

```
ALPS JOBS LAUNCHED ON COMPUTE NODES
Job ID     User       Size   Age              command line
--- ------ --------   -----  ---------------  ----------------------------------
 a  30626  user1          1  1h36m            arps_mpi
 b  30625  user1          1  1h36m            pop.2
 c  30627  user1          1  1h36m            aldh2_hydride
 d  30631  user1          1  1h36m            pop.1
```

```
% xtshowcabs
Compute Processor Allocation Status as of Wed Sep 12 08:09:40 2007

     C0-0     C1-0     C2-0     C3-0     C4-0     C5-0     C6-0     C7-0
  n3 -------- -------- -------- -------- -------- -------- -------- --------
  n2 -------- -------- -------- -------- -------- -------- -------- --------
  n1 -------- -------- -------- -------- -------- -------- -------- --------
c2n0 -------- -------- -------- -------- -------- -------- -------- --------
  n3 -------- -------- -------- -------- -------- -------- -------- --------
  n2 d------- -------- -------- -------- -------- -------- -------- --------
  n1 -------- -------- -------- -------- -------- -------- -------- --------
c1n0 -------- -------- -------- -------- -------- -------- -------- --------
  n3 SSSSSS-- -------- -------- -------- -------- -------- -------- --------
  n2       b- -------- -------- -------- -------- -------- -------- --------
  n1       ac -------- -------- -------- -------- -------- -------- --------
c0n0 SSSSSS-- -------- -------- -------- -------- -------- -------- --------
    s01234567 01234567 01234567 01234567 01234567 01234567 01234567 01234567

Legend:
   nonexistent node                 S  service node
;  free interactive compute CNL     -  free batch compute node CNL
A  allocated, but idle compute node ?  suspect compute node
X  down compute node                Y  down or admindown service node
Z  admindown compute node           R  node is routing

Available compute nodes:       0 interactive,   740 batch

ALPS JOBS LAUNCHED ON COMPUTE NODES
Job ID     User       Size   Age              command line
--- ------ --------   -----  ---------------  ----------------------------------
 a  30626  user1          1  1h40m            arps_mpi
 b  30625  user1          1  1h40m            pop.2
 c  30627  user1          1  1h40m            aldh2_hydride
 d  30631  user1          1  1h40m            pop.1
```

Use `xtshowmesh` on systems with topology class 0 or 4 and `xtshowcabs` on systems with topology class 1, 2, or 3. Contact your system administrator if you do not know the topology class of your system.

Note: If `xtshowcabs` or `xtshowmesh` indicates that no compute nodes have been allocated for interactive processing, you can still run your job interactively by using the PBS Pro `qsub -I` command and then, when your job has been queued, using either the `aprun` or `yod` application launch command.

For more information, see the `xtprocadmin`(1), `xtshowmesh`(1), and `xtshowcabs`(1) man pages.

Running CNL Applications [7]

The aprun utility launches applications on CNL compute nodes. The utility submits applications to the Application Level Placement Scheduler (ALPS) for placement and execution, forwards the user's environment, forwards signals, and manages the stdin, stdout, and stderr streams.

This chapter describes how to run applications interactively on CNL compute nodes and get application status reports. For a description of batch job processing, see Chapter 9, page 67.

7.1 aprun Command

You use the aprun command to specify the resources your application requires, request application placement, and initiate application launch.

The format of the aprun command is:

aprun [-n *pes*] [-N *pes_per_node*] [-d *depth*] [-L *nodes*] [*other arguments*] *executable_name*

where:

aprun **option**	**Description**
-n *pes*	The number of processing elements (PEs) needed for the application. A PE is an instance of an ALPS-launched executable. The -n option applies to both single-core and dual-core systems.
-N *pes_per_node*	The number of PEs per node. The -N option applies only to dual-core systems.
-d *depth*	The number of threads per PE. The default is 1. The -d option applies only to dual-core systems. Compute nodes must have at least *depth* cores.
-L *nodes*	A user-defined placement node list. The node list must contain at least enough nodes to meet the application resource requirements. If the placement node list is too short for the -n, -d, and -N options, a fatal error is produced. See the cnselect(1) man page for details.

You use the `-n` *pes* option to request processing elements (PEs). PEs are instances of the executable.

Note: Verify that you are in a Lustre-mounted directory before using the `aprun` command (see Section 2.4, page 11).

For single-core nodes, ALPS creates `-n` PEs and launches them on `-n` nodes.

For example, the command:

```
% aprun -n 64 ./prog1
```

creates 64 instances of `prog1` and launches them on 64 nodes.

For dual-core nodes, ALPS creates `-n` PEs and uses the `-N` *pes_per_node* value in determining where to place them. Whenever possible, ALPS packs the PEs, using the smallest number of nodes to fulfill the `-n` requirements. If you specify `-N 1`, ALPS assigns one PE per node.

For example, the command:

```
% aprun -n 32 ./prog1
```

creates 32 instances of `prog1` and launches them on both cores of 16 nodes.

In contrast, the command:

```
% aprun -n 32 -N 1 ./prog1
```

creates 32 instances of `prog1` and launches them on one core of 32 nodes. The other 32 cores are unused.

For OpenMP applications, use the `-d` option to specify the depth (number of threads) of each PE. ALPS creates `-n` *pes* instances of the executable, and the executable spawns `depth`-1 additional threads per PE.

For example, the command:

```
% aprun -n 8 -d 2 ./openmp1
```

creates 8 instances of `openmp1` on 8 nodes. Each PE spawns one additional thread.

For examples of CNL applications, see Chapter 13, page 95. For additional information on `aprun`, see the `aprun`(1) man page.

7.2 apstat Command

The apstat command provides status information about reservations, compute resources, and pending and placed applications. The format of the apstat command is:

apstat [-a [*apid* [*apid*...]]] [-n] [-p] [-r] [*other arguments*]

You can use apstat to display the status of all applications (a), specific applications (a *apid*), nodes (n), pending applications (p), and confirmed and claimed reservations (r).

For example:

```
% apstat -a
Placed  Apid  ResId    User   PEs Nodes   Age    Command
        48062    39   user1     2     1   2h39m  test1
        48108  1588   user2     4     1   0h15m  mpi2
        48109  1589   user3     4     1   0h01m  omp1
```

An application's ID (Apid) in the apstat display is also displayed after aprun execution results, such as:

```
% aprun -n 2 -d 2 ./omp1
Hello from rank 0 (thread 0) on nid00540 <-- MASTER
Hello from rank 1 (thread 0) on nid00541 <-- MASTER
Hello from rank 0 (thread 1) on nid00540 <-- slave
Hello from rank 1 (thread 1) on nid00541 <-- slave
Application 48109 resources: utime 0, stime 0%
```

For further information, see the apstat(1) man page.

7.3 cnselect Command

The aprun utility supports manual and automatic node selection. For manual node selection, first use the cnselect command to get a list of compute nodes that meet the criteria you specify. Then use the aprun -L *nodes* option to launch the application. If the number of nodes in the -L *nodes* list is greater than the aprun *n* value, ALPS launches the application on *n* nodes from the -L *nodes* list.

The format of the cnselect command is:

cnselect [-c] [-l] [[-L] *fieldname*|[-e] *expression*]
[*other arguments*]

where:

- `-c` gives a count of the number of nodes rather than a list of the nodes themselves.
- `-l` lists names of fields in the compute nodes attributes database.
- `-L` *fieldname* lists the current possible values for a given field.
- [`-e`] *expression* queries the compute node attributes database.

You can use `cnselect` to get a list of nodes selected by such characteristics as number of cores per node (`coremask`), amount of memory on the node (in megabytes), and processor speed (in megahertz). For example, to run an application on dual-core nodes with 2 GB of memory or more, use:

```
% cnselect availmem .ge. 2000 .and. coremask .gt. 1
44-63,76,82
% aprun -n 16 -L 44-59 ./app1
```

If you do not include `-L` option on the `aprun` command, ALPS automatically places the application per available resources.

7.4 Memory Available to CNL Applications

When running large applications, it is important to understand how much memory will be available per node for your application.

CNL uses approximately 250 MB of memory. The remaining memory is available for the user program executables; user data arrays; the stacks, libraries and buffers; and SHMEM symmetric stack heap. For a node with 2.147 GB of memory, 1.897 GB of memory is available for applications. The default stack size is 16 MB. The memory used for the MPI libraries is approximately 72 MB.

Note: The actual amount of memory CNL uses varies depending on the total amount of memory on the node and the OS services configured for the node.

You can use the aprun -m *size* option to specify the per-PE memory limit. For example, the following aprun command launches program1 on cores 0 and 1 of a compute node with 4 GB of available memory:

```
% aprun -n 2 -N 2 -m2000 ./program1
 hello from pe            0  of            2
 hello from pe            1  of            2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Application 14154 resources: utime 0, stime 0
```

You can change MPI buffer sizes and stack space from the defaults by setting certain environment variables or aprun options. For more details, see the aprun(1) and intro_mpi(3) man pages.

7.5 Launching an MPMD Application

The aprun utility supports multiple-program, multiple-data (MPMD) applications. To run an MPMD application under aprun, use the -n *pes executable1* : -n *pes executable2* : ... format. To communicate with each other, all of the executables share the same MPI_COMM_WORLD process communicator.

This command launches 128 instances of program1 and 256 instances of program2:

```
aprun -n 128 ./program1: -n 256 ./program2
```

7.6 Managing Compute Node Processors from an MPI Program

Programs that use MPI library routines for parallel control and communication should call the MPI_Finalize() routine at the conclusion of the program. This call waits for all processing elements to complete before exiting.

However, if one of the processes fails to call `MPI_Finalize()` for any reason, the program never completes and `aprun` stops responding. There are two ways to prevent this behavior:

- Use the PBS Pro elapsed (wall clock) time limit to terminate the job after a specified time limit (such as `-l walltime=2:00:00`).
- Use the `aprun -t` *`sec`* option to terminate the offending processes. This option specifies the per-process CPU time limit in seconds. A process will terminate only if it reaches the specified amount of CPU time (not wallclock time).

  For example, if you use:

  ```
  % aprun -t 120 ./myprog1
  ```

  and a process consumes more than 2 minutes of CPU time, `aprun` will terminate the application.

7.7 Input and Output Modes under `aprun`

The `aprun` utility handles standard input (`stdin`) on behalf of the user and handles standard output (`stdout`) and standard error messages (`stderr`) for user applications.

For other I/O considerations, see Section 4.2.2, page 27.

7.8 Signal Handling under `aprun`

The `aprun` utility catches and forwards these signals to an application: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGTERM`, `SIGABRT`, `SIGUSR1`, and `SIGUSR2`. The `aprun` utility ignores `SIGPIPE` and `SIGTTIN` signals. All other signals are left at their default behavior and are not forwarded to an application. Those default behaviors cause `aprun` to be terminated, resulting in the application being terminated by a `SIGKILL` signal.

Running Catamount Applications [8]

The yod utility launches applications on Catamount compute nodes. When you start a yod process, the application launcher coordinates with the Compute Processor Allocator (CPA) to allocate nodes for the application and then uses Process Control Threads (PCTs) to transfer the executable to the compute nodes. While the application is running, yod provides I/O services for the application, propagates signals, and participates in cleanup when the application terminates.

This chapter describes how to run applications interactively on Catamount compute nodes. For a description of batch job processing, see Chapter 9, page 67.

8.1 yod Command

When launching an application with the yod command, you can specify the number of processors to allocate to the application.

The format of the yod command is:

```
% yod -sz n [other arguments] executable_name
```

where *n* is the number of processors on which the application will run.

The yod -sz, -size, and -np options are synonymous.

The following paragraphs describe the differences in the way processors are allocated on single-core and dual-core processor systems.

- Running applications on single-core processor systems

  On single-core processor systems, each compute node has one single-core AMD Opteron processor. Applications are allocated -sz nodes.

  For example, the command:

  ```
  % yod -sz 6 prog1
  ```

  launches prog1 on six nodes.

  Single-core processing is the default. However, sites can change the default to dual-core processor mode. Use -SN if the default is dual-core processor mode and you want to run applications in single-core processor mode.

Note: The `yod -VN` option turns on virtual node processing mode. The `yod` utility runs the program on both cores of a dual-core processor. If you use the `-VN` option on a single-core system, the application load will fail.

- Running applications on dual-core processor systems

  On dual-core processor systems, each compute node has one dual-core AMD Opteron processor. The processors are managed by the Catamount Virtual Node (CVN) kernel. To launch an application, you must include the `-VN` option on the `yod` command unless your site has changed the default.

  On a dual-core system, if you do not include the `-VN` option, your program will run on one core per node, with the other core idle. You may do this if you must use all the memory on a node for each processing element or if you want the fastest possible run time and do not mind letting the second core on each node sit idle.

8.2 `cnselect` Command

The `yod` utility supports automatic and manual node selection. To use manual node selection, first use the `cnselect` command to get a list of compute nodes that meet the criteria you specify. Then use the `yod -list` *processor-list* option to launch the application. If the number of nodes in the list is greater than the `-sz` *n* value, `yod` selects *n* of the *processor-list* nodes on which to launch the application.

The format of the `cnselect` command is:

```
cnselect [-c] [-l] [[-L] fieldname|[-e] expression]
[other arguments]
```

where:

- `-c` gives a count of the number of nodes rather than a list of the nodes themselves.
- `-l` lists names of fields in the compute nodes attributes database.
- [`-L`] *fieldname* lists the current possible values for a given field.
- [`-e`] *expression* queries the compute node attributes database.

You can use `cnselect` to get a list of nodes selected by such characteristics as number of cores per node (`coremask`), available memory (in megabytes), and processor speed (in megahertz). For example, to run an application on dual-core nodes with 2 GB of memory or more, use:

```
% cnselect -y availmem .ge. 2000 .and. coremask .gt. 1
44..63,76,82
% yod -VN -sz 16 -list 44..59 ./app1
```

Note: When using `cnselect` with `yod`, you need to include the `-y` option on the `cnselect` command. This option causes `cnselect` to list ranges of nodes in `yod` format (`n..n`).

If you do not include `-list` option, `yod` automatically places the application per available resources.

8.3 Memory Available to Catamount Applications

When running large applications on a dual-core processor system, it is important to understand how much memory will be available per node for your job.

If you are running in single-core mode on a dual-core system, Catamount (the kernel plus the process control thread (PCT)) uses approximately 120 MB of memory. The remaining memory is available for the user program executable, user data arrays, the stack, libraries and buffers, and SHMEM symmetric stack heap.

For example, on a node with 2.147 GB of memory, memory is allocated as follows:

Catamount	120 MB (approximate)
Executable, data arrays, stack, libraries and buffers, SHMEM symmetric stack heap	2027 MB (approximate)

If you are running in dual-core mode, Catamount uses approximately 120 MB of memory (the same as for single-core mode). The PCT divides the remaining memory in two, allocating half to each core. The memory allocated to each core is available for the user executable, user data arrays, stack, libraries and buffers, and SHMEM symmetric stack heap.

For example, on a node with 2.147 GB of memory, memory is allocated as follows:

Catamount	120 MB (approximate)
Executable, data arrays, stack, libraries and buffers, SHMEM symmetric stack heap for core 0	1013 MB (approximate)
Executable, data arrays, stack, libraries and buffers, SHMEM symmetric stack heap for core 1	1013 MB (approximate)

The default stack size is 16 MB.

The memory used for the Lustre and MPI libraries is as follows:

Lustre library	17 MB (approximate)
MPI library and default buffer	72 MB (approximate)

You can change MPI buffer sizes and stack space from the defaults by setting certain environment variables or `yod` options. For more details, see the `yod`(1) and `intro_mpi` `man`(3) pages.

8.4 Launching an MPMD Application

The `yod` utility supports multiple-program, multiple-data (MPMD) applications of up to 32 separate executable images. To run an MPMD application under `yod`, first create a *loadfile* where each line in the file is the `yod` command for one executable image. To communicate with each other, all of the executable images launched in a loadfile share the same `MPI_COMM_WORLD` process communicator.

The following yod options are valid within a loadfile:

-heap *size*

Specifies the number of bytes to reserve for the heap. The minimum value of *size* is 16 MB. On dual-core systems, each core is allocated *size* bytes.

-list *processor-list*

Lists the candidate compute nodes on which to run the application, such as: -list 42,58,64..100,150..200. Use the cnselect command with the -y option to generate the list. See the cnselect(1) man page for details.

-shmem *size*

Specifies the number of bytes to reserve for the symmetric heap for the SHMEM library. The heap size is rounded up in order to address physical page boundary issues. The minimum value of size is 2 MB. On dual-core systems, each core is allocated *size* bytes.

-size|-sz|-np *n*

Specifies the number of processors on which to run the application. In SN mode, -size *n* is the number of nodes. In VN mode, -size *n* is the number of cores. You can use the -size option in conjunction with the -list option to launch an application on a subset of the -list *processor-list* nodes.

-stack *size*

Specifies the number of bytes to reserve for the stack. On dual-core systems, each core is allocated *size* bytes.

This loadfile script launches program1 on 128 nodes and program2 on 256 nodes:

```
#loadfile
yod -sz 128 program1
yod -sz 256 program2
```

To launch the application, use:

```
% yod -F loadfile
```

8.5 Managing Compute Node Processors from an MPI Program

Programs that use MPI library routines for parallel control and communication should call the `MPI_Finalize()` routine at the conclusion of the program. This call waits for all processing elements to complete before exiting. However, if one of the processes fails to start or stop for any reason, the program never completes and `yod` stops responding. To prevent this behavior, use the `yod -tlimit` option to terminate the application after a specified number of seconds. For example,

```
% yod -tlimit 30K myprog1
```

terminates all processes remaining after 30K (30 * 1024) seconds so that `MPI_Finalize()` can complete. You can also use the environment variable `YOD_TIME_LIMIT`. The time limit specified on the command line overrides the value specified by the environment variable.

8.6 Input and Out Modes under yod

All standard I/O requests are funneled through `yod`. The `yod` utility handles standard input (`stdin`) on behalf of the user and handles standard output (`stdout`) and standard error messages (`stdout`) for user applications.

For other I/O considerations, see Section 4.3.2, page 31.

8.7 Signal Handling under yod

The `yod` utility uses two signal handlers, one for the load sequence and one for application execution. During the load operation, any signal sent to `yod` during the load operation terminates the operation. After the load is completed and all nodes of the application have signed in with `yod`, the second signal handler takes over.

During the execution of a program, `yod` interprets most signals as being intended for itself rather than the application. The only signals propagated to the application are `SIGUSR1`, `SIGUSR2`, and `SIGTERM`. All other signals effectively terminate the running application. The application can ignore the signals that `yod` passes along to it; `SIGTERM`, for example, does not necessarily terminate an application. However, a `SIGINT` delivered to `yod` initiates a forced termination of the application.

8.8 Associating a Project or Task with a Job Launch

Use the `-Account "`*project task*`"` or `-A "`*project task*`"` `yod` option or the `-A "`*project task*`"` `qsub` option to associate a job launch with a particular project and task. Use double quotes around the string that specifies the project and, optionally, task values. For example:

```
% yod -Account "grid_test_1234 task1" -sz 16 myapp123
```

You can also use the environment variable `XT_ACCOUNT="`*project task*`"` to specify account information. The `-Account` or `-A` option overrides the environment variable.

If `yod` is invoked from a batch job, the `qsub -A` account information takes precedence; `yod` writes a warning message to `stderr` in this case.

Using PBS Pro [9]

Your Cray XT series Programming Environment may include the optional PBS Pro batch scheduling software package from Altair Grid Technologies. This section provides an overview of job processing under PBS Pro.

The Cray XT series system can be configured with a given number of interactive job processors and a given number of batch processors. A job that is submitted as a batch process can use only the processors that have been allocated to the batch subsystem. If a job requires more processors than have been allocated for batch processing, it remains in the batch queue but never exits.

> **Note:** At any time, the system administrator can change the designation of any node from interactive to batch or vice versa. However, this does not affect jobs already running on those nodes. It applies only to jobs that are in the queue and to subsequent jobs.

The basic process for creating and running batch jobs is to create a PBS Pro job script that includes `aprun` or `yod` commands and then use the PBS Pro `qsub` command to run the script.

9.1 Creating Job Scripts

A job script may consist of directives, comments, and executable statements. A PBS Pro directive provides a way to specify job attributes apart from the command line options:

```
#PBS -N job_name
#PBS -l resource_type=specification
#
command
command
...
```

PBS Pro provides a number of *resource_type* options for specifying, allocating, and scheduling compute node resources, such as `mppwidth` (number of processing elements), `mppdepth` (number of threads), and `mppnodes` (manual node placement list). See Table 6, page 68, Table 7, page 69, and the `pbs_resources`(7B) man page for details.

9.2 Submitting Batch Jobs

To submit a job to the batch scheduler, use the following commands:

```
% module load pbs
% qsub [-l resource_type=specification] jobscript
```

where *jobscript* is the name of a job script that includes one or more `aprun` or `yod` commands.

The `qsub` command scans the lines of the script file for directives. An initial line in the script that begins with the characters `#!` or the character `:` is ignored and scanning starts at the next line. Scanning continues until the first executable line (that is, a line that is not blank, not a directive line, nor a line whose first non-white-space character is #). If directives occur on subsequent lines, they are ignored.

If a `qsub` option is present in both a directive and on the command line, the command line takes precedence. If an option is present in a directive and not on the command line, that option and its argument, if any, are processed as if you included them on the command line.

9.2.1 Using aprun with qsub

For CNL jobs, the `qsub -l` *resource_type=specification* options and `aprun` options are defined as follows:

Table 6. `aprun` versus `qsub` Options

`aprun` **option**	`qsub -l` **option**	**Description**
`-n 4`	`-l mppwidth=4`	Width (number of PEs)
`-d 2`	`-l mppdepth=2`	Depth (number of OpenMP threads)
`-N 1`	`-l mppnppn=1`	Number of PEs per node
`-L 5,6,7`	`-l mppnodes=\"5,6,7\"`	Node List
`-m 1000m`	`-l mppmem=1000mb`	Memory per PE

For examples of batch jobs that use `aprun`, see Chapter 13, page 95.

9.2.2 Using yod with qsub

On a single-core system, the PBS Pro *mppwidth* parameter is equivalent to the `yod sz` option.

On a dual-core system, the PBS Pro *mppwidth* parameter is **not** equivalent to the `yod sz` option. The PBS Pro *mppwidth* parameter refers to the number of **nodes** to be allocated for a job. The `yod sz` option refers to the number of **cores** to be allocated for a job (two cores per node).

For example, the following commands:

```
% qsub -I -V -l mppwidth=6
% yod -size 12 -VN prog1
```

allocate 6 nodes to the job and launch `prog1` on both cores of each of the 6 nodes.

For Catamount jobs, the `qsub -l` *resource_type=specification* options and `yod` options are defined as follows:

Table 7. `yod` versus `qsub` Options

`yod` **option**	`qsub -l` **option**	**Description**
`-sz 4`	`-l mppwidth=4`	Number of processors (single core)
`-VN -sz 8`	`-l mppwidth=4`	Number of processors (dual core)
`-list 5,6,7`	`-l mppnodes=\"5,6,7\"`	Node List

For examples of batch jobs that use `yod`, see Chapter 14, page 133.

9.3 Terminating Failing Processes in an MPI Program

Jobs that use MPI library routines for parallel control and communication should call the `MPI_Finalize()` routine at the conclusion of the program. This call waits for all processing elements to complete before exiting. However, if one of the processes fails to start or stop for any reason, the program never completes and `aprun` or `yod` stops responding. To prevent this behavior, use the PBS Pro time limit to terminate remaining processes so that `MPI_Finalize()` can complete.

9.4 Getting Jobs Status

The `qstat` command displays the following information about all jobs currently running under PBS Pro:

- The job identifier (`Job id`) assigned by PBS Pro
- The job name (`Name`) given by the submitter
- The job owner (`User`)
- CPU time used (`Time Use`)
- The job state (`S`): whether job is exiting (`E`), held (`H`), in the queue (`Q`), running (`R`), suspended (`S`), being moved to a new location (`T`), or waiting for its execution time (`W`)
- The queue (`Queue`) in which the job resides

For example:

```
% qstat
Job id Name             User             Time Use S Queue
------ ---------------- ---------------- -------- - -----
84.nid00003 test_ost4_7 usera            03:36:23 R workq
33.nid00003 run.pbs     userb            00:04:45 R workq
34.nid00003 run.pbs     userb            00:04:45 R workq
35.nid00003 STDIN       userc            00:03:10 R workq
```

If the `-a` option is used, queue information is displayed in the alternative format.

```
% qstat -a
nid00003:
                                                  Time In Req'd  Req'd   Elap
Job ID Username Queue    Jobname    SessID  Queue  Nodes  Time  S Time
------ -------- -------- ---------- ------ ------- ------ ----- - -----
163484 usera    workq    test_ost4_   9143  003:48     64   --  R 03:47
163533 userb    workq    run.pbs     15040  000:48     64 00:30 R 00:15
163534 userb    workq    run.pbs     15045  000:48     64 00:30 R 00:15
163536 userc    workq    STDIN       15198  000:10      5   --  R 00:09

Total generic compute nodes allocated: 197
```

For details, see the `qstat`(1B) man page.

9.5 Removing a Job from the Queue

The `qdel` command removes a PBS Pro batch job from the queue. As a user, you can remove any batch job for which you are the owner. Jobs are removed from the queue in the order they are presented to `qdel`. For more information, see the `qdel`(1B) man page and the *PBS Pro User Guide*.

Debugging an Application [10]

This chapter describes some of the debugging options that are native to the Cray XT series Programming Environment, as well as the optional TotalView debugging software package from TotalView Technologies, LLC and the GNU `gdb` debugger.

10.1 Troubleshooting Catamount Application Failures

The `yod` utility provides rudimentary diagnostics for a subset of compute node operating system calls. The subset consists of the following system calls, which perform remote procedure calls (RPCs) to `yod`:

Table 8. RPCs to `yod`

`chmod`	`fstatfs`	`mkdir`	`rmdir`	`symlink`
`chown`	`fsync`	`open`	`setegid`	`sync`
`close`	`ftruncate`	`pread`	`seteuid`	`truncate`
`exit`	`getdirentries`	`pwrite`	`setgid`	`umask`
`fchmod`	`link`	`read`	`setuid`	`unlink`
`fchown`	`lseek`	`readlink`	`stat`	`utimes`
`fstat`	`lstat`	`rename`	`statfs`	`write`

System calls that are performed solely by Catamount do not show up in the diagnostic output.

There are two ways to enable this feature:

- Invoke `yod` with the `-strace` option.
- Set `YOD_STRACE=1` in your shell environment.

Note: In this context the term *strace* is a misnomer. The `yod` utility does not provide the UNIX-like `strace()` function. Enabling `strace` turns on diagnostic output generated by the RPC library, which `yod` uses to service the system calls listed previously. The I/O-related system calls are for non-parallel file systems.

The yod command also enables you to get trace reports about memory allocation and deallocation. The -tracemalloc option provides rudimentary diagnostics for malloc() and free() calls. This information can help you pinpoint memory leaks and determine if using the GNU malloc library would be beneficial. For further information about the GNU malloc library, see Appendix B, page 187.

10.2 Using the TotalView Debugger

Cray XT series systems support the TotalView debugger. TotalView is an optional product that provides source-level debugging of applications running on multiple compute nodes. TotalView is compatible with the PGI, GCC, and PathScale compilers.

TotalView:

- Provides both a graphical user interface and a command-line interface (with command-line help).
- Supports the x86-64 Assembler.
- Supports programs written in mixed languages.
- Supports debugging of up to 4096 compute node processes.
- Supports watchpoints.
- Provides a memory debugger.

TotalView typically is run interactively. If your site has not designated any compute nodes for interactive processing, use the PBS Pro qsub -I interactive mode described in Chapter 9, page 67.

For further information about the TotalView graphical and command line interfaces, see the totalview(1) man page. For further information about TotalView, including details about running on a Cray XT series system, see http://www.totalviewtech.com/Documentation.

10.2.1 Debugging an Application

To debug a CNL application, use this command format to launch an instance of aprun, which in turn launches the application *executable_name*:

```
% totalview aprun -a [other_aprun_arguments] ./executable_name
```

Note: The `-a` option is a TotalView option indicating that the arguments that follow apply to `aprun`. If you want to use the `aprun -a` *`arch`* option, you need to include a second `-a`, as in:

```
% totalview aprun -a -a xt -n 2 ./a.out
```

For example, to debug application `xt1`, use:

```
% totalview aprun -a -n 2 ./xt1
```

The TotalView Root and Process windows appear.

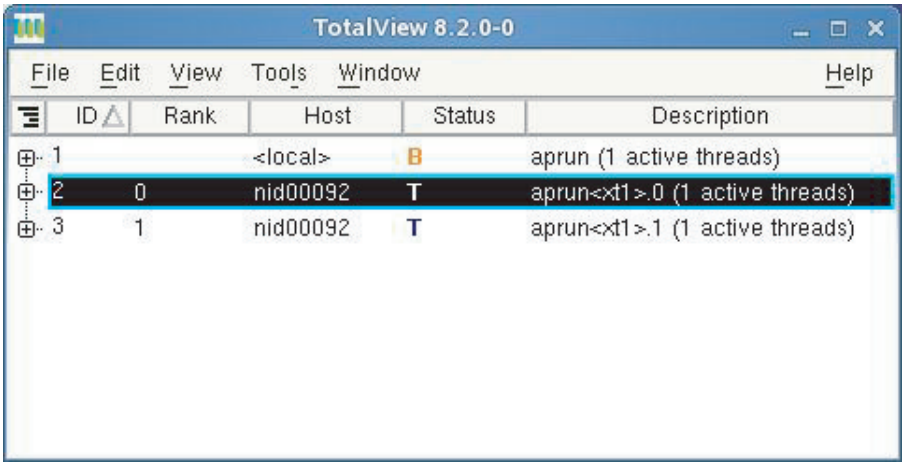

Figure 1. TotalView Root Window

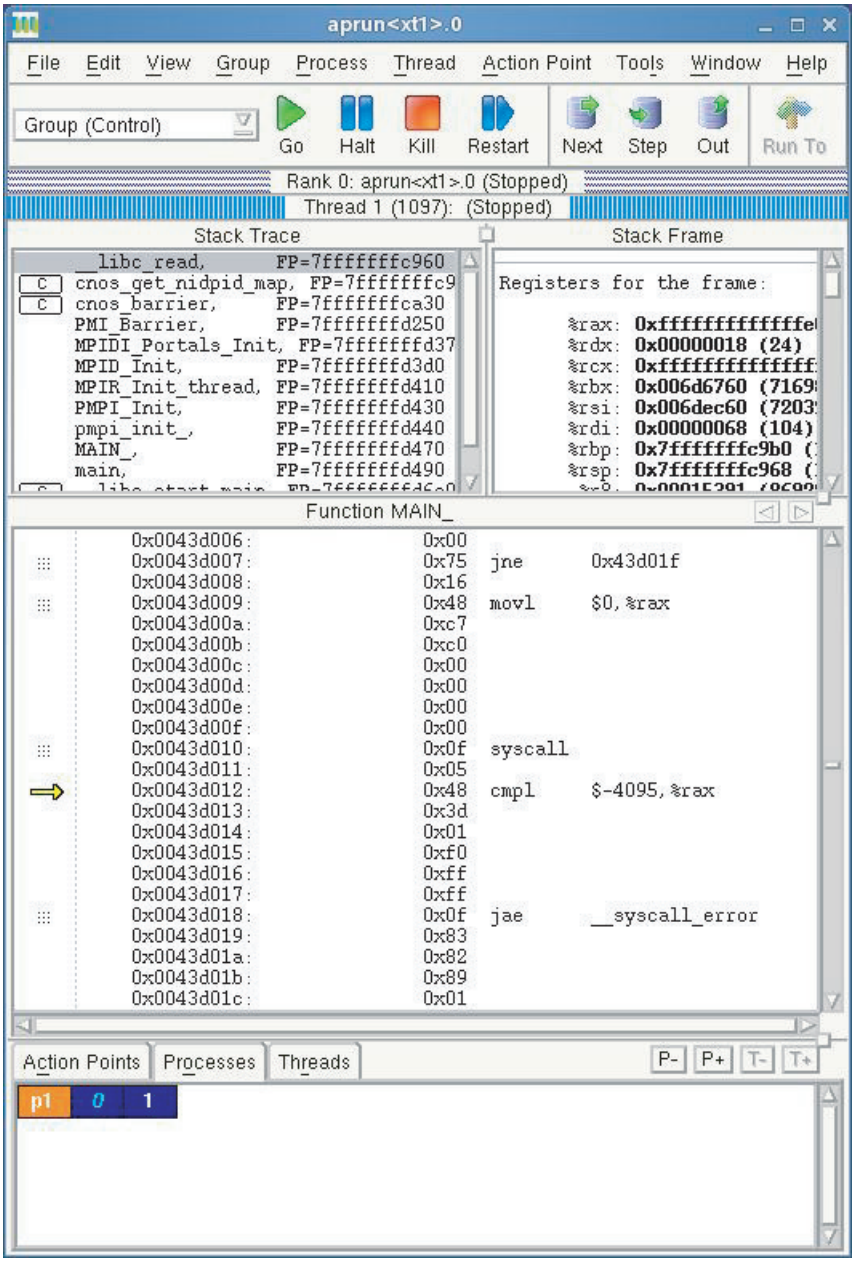

Figure 2. TotalView Process Window

To debug a Catamount application, substitute `yod` for `aprun` in the `totalview` command.

10.2.2 Debugging a Core File

To debug a core file, from the Process window **File** menu, select **New Program**. A New Program window appears. Click the **Open a core file** icon. Under the **Program** tab, specify the application name in the **Program:** field and the core file name in the **Core file:** field. Click **OK**.

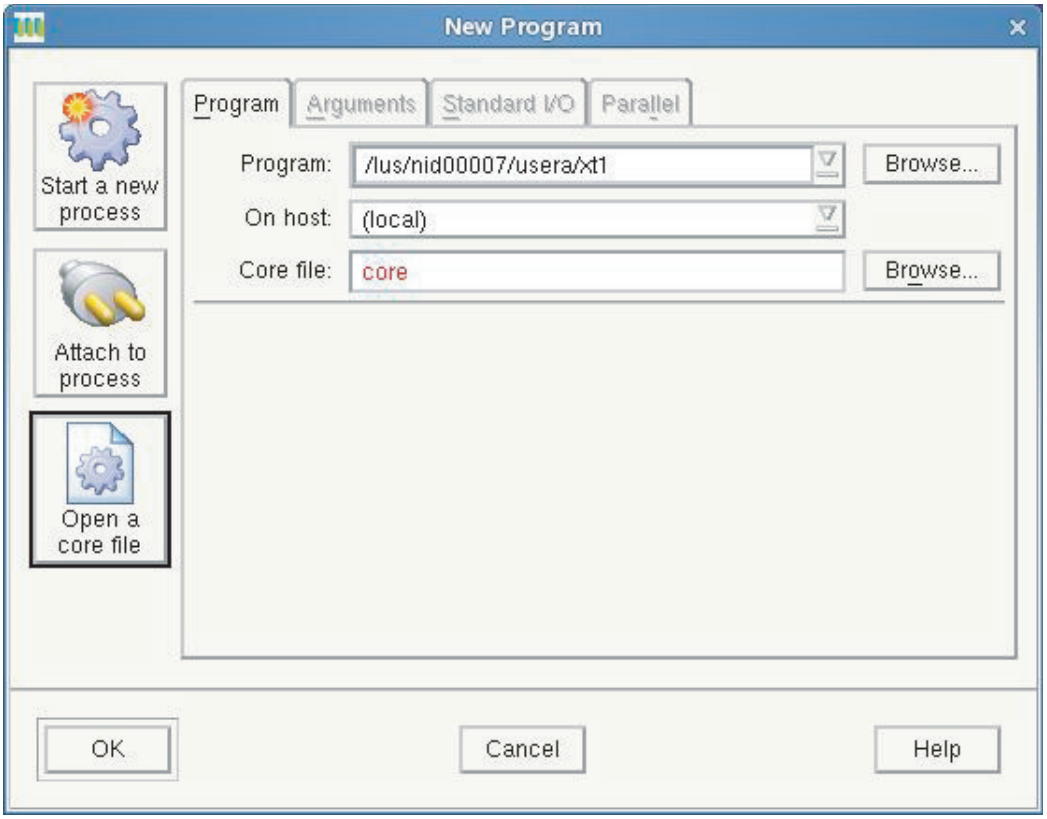

Figure 3. Debugging a Core File

10.2.3 Attaching to a Running Process

To attach TotalView to a running process, you must be logged in to the same login node that you used to launch the process, and you must attach to the instance of `aprun` that was used to launch the process, rather than to the process itself. To do so, follow these steps:

1. Launch TotalView:

   ```
   % totalview
   ```

2. In the New Program window, click the **Attach to an existing process** icon. The list of processes currently running displays.

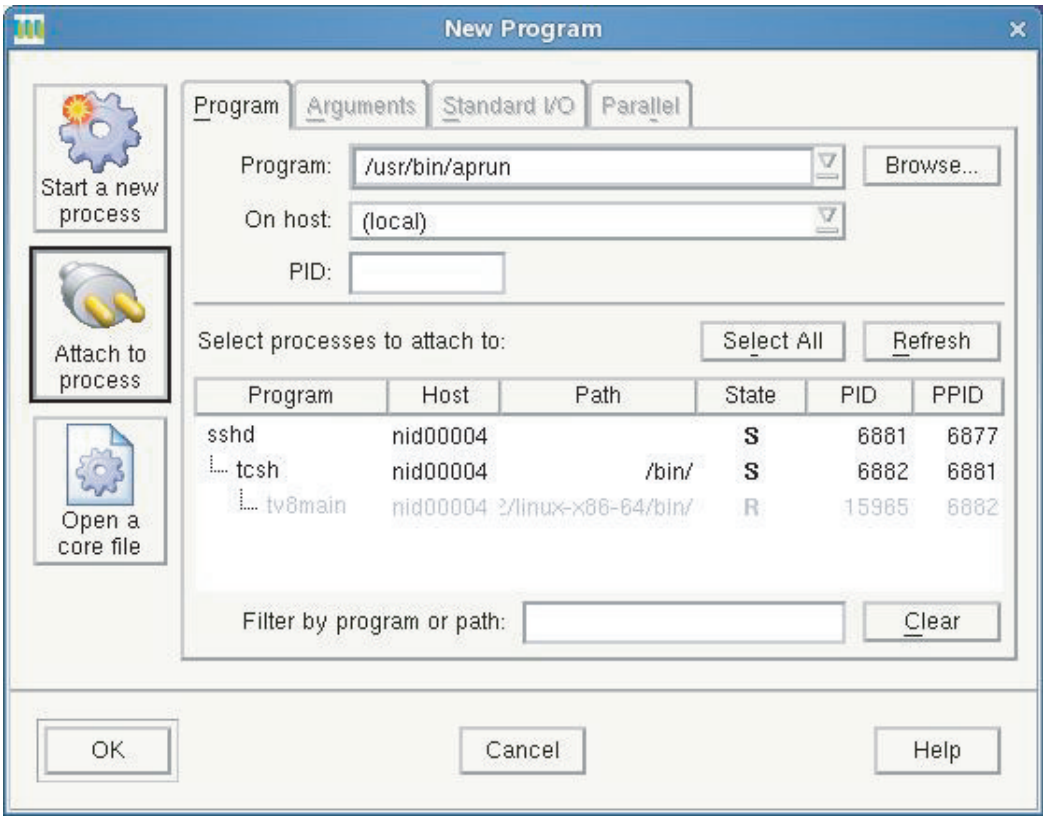

Figure 4. Attaching to a Running Process

3. Select the instance of `aprun` you want, and click **OK**. TotalView displays a Process Window showing both `aprun` and the program threads that were launched using that instance of `aprun`.

10.2.4 Altering Standard I/O

To change the names of the files to which TotalView will write or from which TotalView will read, Launch the program using TotalView. Do not specify the `stdin` file at this time. Use:

```
% totalview aprun -a -n pes program_name
```

The TotalView Root and Process windows display. In the Process window under the File menu, select New Program. The New Program window displays. Select the Standard I/O tab. The Standard Input, Standard Output, and Standard Error fields are displayed.

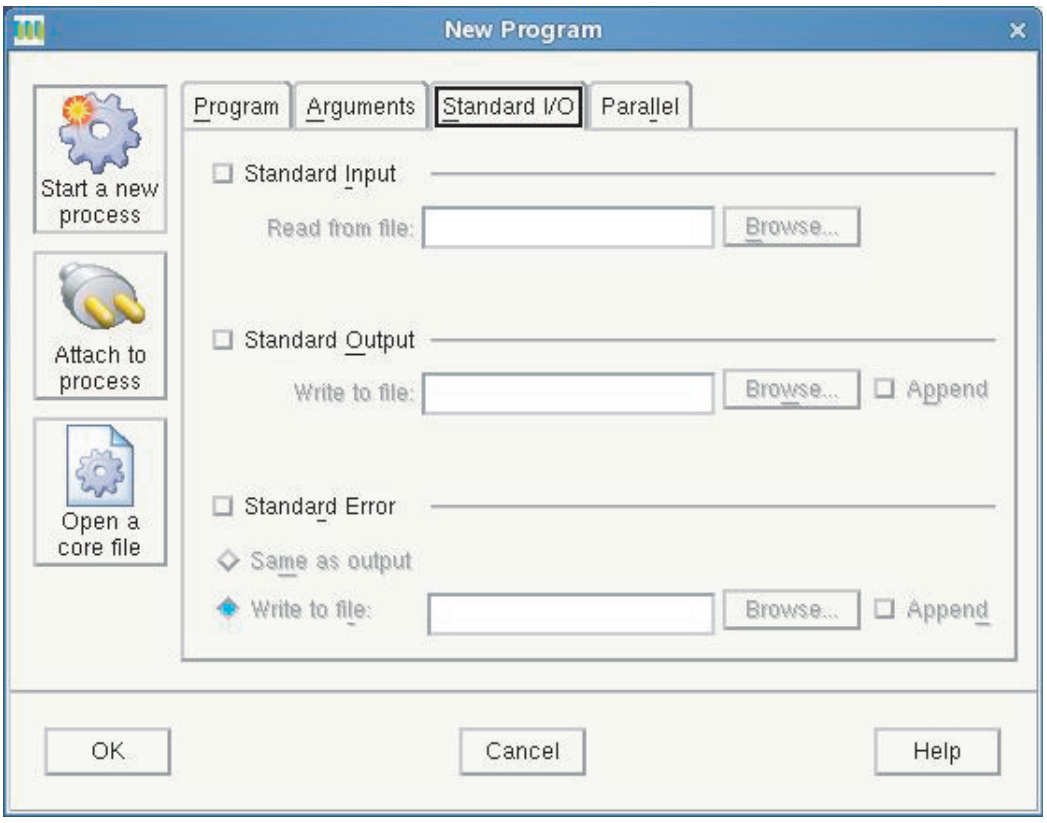

Figure 5. Altering Standard I/O

Type the file name for **Standard Input**,**Standard Output**, or **Standard Error** field, specify the desired file name, and click the **OK** button.

On the main TotalView window, click the **Go** button to begin program execution.

10.2.5 TotalView Limitations for Cray XT Series Systems

The TotalView debugging suite for the Cray XT series system differs in functionality from the standard TotalView implementation in the following ways:

- The TotalView Visualizer is not included.
- Debugging multiple threads on compute nodes is not supported.
- Debugging `MPI_Spawn()`, OpenMP, or Cray SHMEM programs is not supported.
- Compiled `EVAL` points and expressions are not supported.
- Type transformations for the PGI C++ compiler standard template library collection classes are not supported.
- Exception handling for the PGI C++ compiler run time library is not supported.
- Spawning a process onto the compute processors is not supported.
- Machine partitioning schemes, gang scheduling, or batch systems are not supported.

In some cases, TotalView functionality is limited because CNL or Catamount does not support the feature in the user program.

10.3 Using the GNU gdb Debugger

Cray XT series supports the GNU Project debugger, gdb, for single-process debugging on Catamount compute nodes; gdb is not supported for CNL compute nodes.

Use the `cc`, `CC`, `ftn`, or `f77 -g` debug option to generate debugging information. This information describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

For an example showing how to use `xtgdb` to set breakpoints in a single-process job, see Example 38, page 154.

For details, see the `xtgdb`(1), `cc`(1), `CC`(1), `f77`(1), and `ftn`(1) man pages.

Performance Analysis [11]

This chapter describes the Cray XT series performance analysis tools.

11.1 Using the Performance API

The Performance API (PAPI) is a standard API for accessing microprocessor registers that count events or occurrences of specific signals related to the processor's function. By monitoring these events, you can determine the extent to which your code efficiently maps to the underlying architecture.

PAPI provides two interfaces to the counter hardware:

- A high-level interface for basic measurements
- A fully programmable, low-level interface for users with more sophisticated needs

PAPI supports multiplexing under CNL. Although it is also supported under Catamount, the long time slice (~1 second) for each set of independent counters makes it impractical to use except for very long running programs.

The `pat_build` utility does not allow you to instrument a program that is also using the PAPI interface directly or indirectly (via `libhwpc`).

To use PAPI, you must load the PAPI module.

For CNL applications, use:

```
% module load papi-cnl
```

For Catamount applications, use:

```
% module load papi
```

For more information about PAPI, see `http://icl.cs.utk.edu/papi/`.

11.1.1 Using the High-level PAPI Interface

The high-level interface provides the ability to start, stop, and read specific events, one at a time. For an example of a CNL application using the PAPI high-level interface, see Example 17, page 114. For an example of a Catamount application using the PAPI high-level interface, see Example 39, page 155.

11.1.2 Using the Low-level PAPI Interface

The low-level PAPI interface deals with hardware events in groups called *event sets*. An event set maps the hardware counters available on the system to a set of predefined events, called *presets*. The event set reflects how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level-1 cache misses can reveal poor locality and memory management.

Event sets are fully programmable and have features such as guaranteed thread safety, writing of counter values, multiplexing, and notification on threshold crossing, as well as processor-specific features. For the list of predefined event sets, see the `hwpc`(3) man page.

For an example of a CNL application using the PAPI low-level interface, see Example 18, page 115. For an example of a Catamount application using the PAPI low-level interface, see Example 40, page 156.

For information about constructing an event set, see the *PAPI User Guide* and the *PAPI Programmer's Reference* manual.

For a list of supported hardware counter presets from which to construct an event set, see Appendix C, page 193.

11.2 Using the Cray Performance Analysis Tool

The Cray Performance Analysis Tool (CrayPat) helps you analyze the performance of programs. To use it:

1. Load the `craypat` module:

   ```
   % module load craypat
   ```

   Note: You must load the `craypat` module before building even the uninstrumented version of the application.

2. Compile and link your application.

   Note: All executable programs previously created with the CrayPat 3.1 module must be relinked in order to be instrumented with CrayPat 3.2. The `pat_build` utility in CrayPat 3.2 will not instrument executable files linked with the CrayPat 3.1 module loaded.

3. Use the `pat_build` command to create an instrumented version of the application, specifying the functions to be traced through options such as `-u` and `-g mpi`.

4. Set any relevant environment variables, such as:

   - `setenv PAT_RT_HWPC 1`, which specifies the first of the nine predefined sets of hardware counter events.
   - `setenv PAT_RT_SUMMARY 0`, which specifies a full-trace data file rather than a summary. Such a file can be very large but is needed to view behavior over time with Cray Apprentice2.
   - `setenv PAT_BUILD_ASYNC 1`, which enables you to instrument a program for a sampling experiment.
   - `setenv PAT_RT_EXPFILE_DIR` *dir*, which enables you to specify a directory into which the experiment data files will be written, instead of the current working directory. If a single data file is written, its default root name is the name of the instrumented program followed by the plus sign (+), the process ID, and one or more key letters indicating the type of the experiment (such as `program1+pat+3820tdt`). If there is a data file from each process, they are written into a subdirectory with that name. For a large number of processes, it may be necessary that `PAT_RT_EXPFILE_MAX` be set to 0 or the number of processes and that `PAT_RT_EXPFILE_DIR` be set to a directory in a Lustre file system (if the instrumented program is not invoked in such a directory). The default for a multi-PE program is to write a single data file.

5. Execute the instrumented program.

6. Use `pat_report` on the resulting data file to generate a report. The default report is a sample by function, but alternative views can be specified through options such as:

   - `-O calltree`
   - `-O callers`
   - `-O load_balance`

   The `-s pe=...` option overrides the way that per-PE data is shown in default tables and in tables specified using the `-O` option. For details, see the `pat_report`(1) man page.

These steps are illustrated in the example CrayPat programs (see Chapter 13, page 95 and Chapter 14, page 133). For more information, see the man pages and the interactive `pat_help` utility.

Note: CrayPat does not support the PathScale `-fb-create`, `-fb-phase`, or `-pg` compiler options.

For more information about using CrayPat, see the *Using Cray Performance Analysis Tools* manual, the `craypat`(1) man page, and run the `pat_help` utility. For more information about PAPI HWPC, see Appendix C, page 193, the `hwpc`(3) man page, and the PAPI website at `http://icl.cs.utk.edu/papi/`.

11.2.1 Tracing and Sampling Experiments

CrayPat supports two types of experiments: tracing and sampling.

Tracing counts an event, such as the number of times an MPI call is executed. When tracing experiments are done, selected function entry points are traced and produce a data record in the run time experiment data file, if the function is executed. The following categories of function entry points can be traced:

- System calls
- I/O (formatted and buffered or system calls)
- Math (see `math.h`)
- MPI
- SHMEM
- Dynamic heap memory
- BLAS
- LAPACK
- Pthreads (not supported on Catamount)

Note: Only true function calls can be traced. Function calls that are inlined by the compiler cannot be traced.

Sampling experiments capture values from the call stack or the program counter at specified intervals or when a specified counter overflows. (Sampling experiments are also referred to as asynchronous experiments).

Supported sampling functions are:

- `samp_pc_prof`, which provides the total user time and system time consumed by a program and its functions (not supported on Catamount).
- `samp_pc_time`, which samples the program counter at a given time interval. This returns the total program time and the absolute and relative times each program counter was recorded.
- `samp_pc_ovfl`, which samples the program counter at a given overflow of a hardware performance counter.
- `samp_cs_time`, which samples the call stack at a given time interval and returns the total program time and the absolute and relative times each call stack counter was recorded (otherwise identical to the `samp_pc_time` experiment).
- `samp_cs_ovfl`, which samples the call stack at a given overflow of a hardware performance counter (otherwise identical to the `samp_pc_ovfl` experiment).
- `samp_ru_time`, which samples system resources at a given time interval (otherwise identical to the `samp_pc_time` experiment).
- `samp_ru_ovfl`, which samples system resources at a given overflow of a hardware performance counter (otherwise identical to the `samp_pc_ovfl` experiment.)
- `samp_heap_time`, which samples dynamic heap memory management statistics at a given time interval (otherwise identical to the `samp_pc_time` experiment).
- `samp_heap_ovfl`, which samples dynamic heap memory management statistics at a given overflow of a hardware performance counter (otherwise identical to the `samp_pc_ovfl` experiment).

Note: Hardware counter information cannot be collected during any type of sampling on a Catamount system and cannot be collected during sampling by overflow on a CNL system. Recommended practice is to use sampling to obtain a profile and then trace the functions of interest to obtain hardware counter information for them.

11.3 Using Cray Apprentice2

Cray Apprentice2 is a performance data visualization tool. You can run Cray Apprentice2 on a Cray XT series system or Cray Apprentice2 Desktop on a standalone Linux machine. After you have used `pat_build` to instrument a program for a performance analysis experiment, executed the instrumented program, and used `pat_report` to convert the resulting data file to a Cray Apprentice2 data format, you can use Cray Apprentice2 to explore the experiment data file and generate a variety of interactive graphical reports.

To run Cray Apprentice2, load the Cray Apprentice2 module, run `pat_report`, then use the `app2` command to launch Cray Apprentice2:

```
% module load apprentice2
% app2 [--limit tag_count | --limit_per_pe tag_count] [data_files]
```

Use the `pat_report -f ap2` option to specify the data file type.

To create a graphical representation of a CrayPat report, use an experiment file to generate a report in XML format.

For example, using experiment file `program1+pat+2511td`, generate a report in XML format (note the inclusion of the `-f ap2` option):

```
% module load apprentice2
% pat_report -f ap2 program1+pat+2511td
Output redirected to:  program1+pat+2511td.ap2
```

Run Cray Apprentice2:

```
% app2 program1+pat+2511td.ap2
```

Cray Apprentice2 displays `pat_report` data in graphical form. This example shows the Function display option:

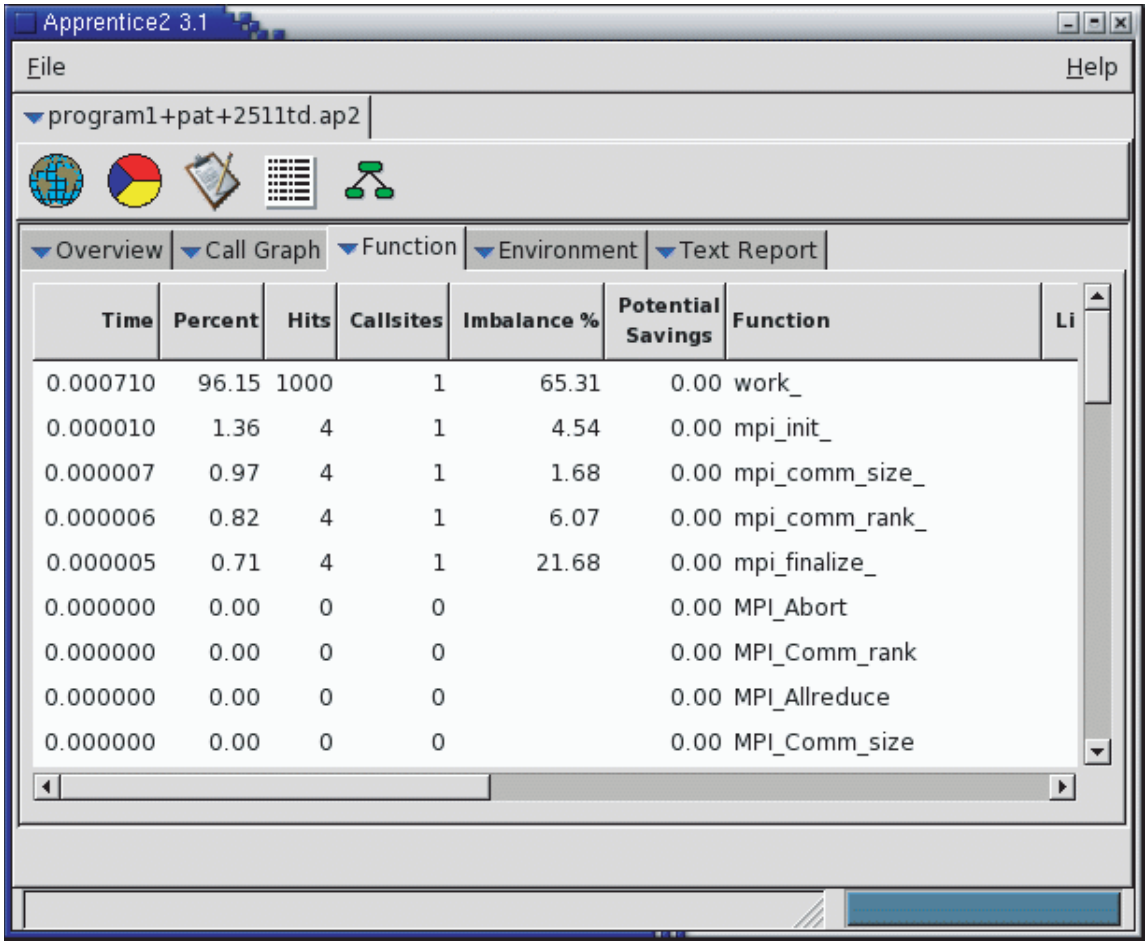

Figure 6. Cray Apprentice2 Function Display

For more information about using Cray Apprentice2, see the Cray Apprentice2 online help system and the `app2`(1) and `pat_report`(1) man pages.

Optimization [12]

12.1 Using Compiler Optimization Options

After you have compiled and debugged your code and analyzed its performance, you can use a number of techniques to optimize performance. For details about compiler optimization and optimization reporting options, see the *PGI User's Guide*, the *Using the GNU Compiler Collection (GCC)* manual, or the *QLogic PathScale Compiler Suite User Guide*.

Optimization can produce code that is more efficient and runs significantly faster than code that is not optimized. Optimization can be performed at the compilation unit level through compiler driver options or to selected portions of code through the use of directives or pragmas. Optimization may increase compilation time and may make debugging difficult. It is best to use performance analysis data to isolate the portions of code where optimization would provide the greatest benefits.

In the following example, a Fortran matrix multiply subroutine is optimized. The compiler driver option generates an optimization report.

Source code of `matrix_multiply.f90`:

```
subroutine mxm(x,y,z,m,n)
real*8 x(m,n), y(m,n), z(n,n)

do k = 1,n
  do j = 1,n
    do i = 1,m
      x(i,j) = x(i,j) + y(i,k)*z(k,j)
    enddo
  enddo
enddo

end
```

PGI Fortran compiler command:

```
% ftn -c -fast -Mvectsse -Minfo matrix_multiply.f90
```

Optimization report:

```
mxm:
     4, Interchange produces reordered loop nest: 5, 4, 6
     6, Generated 3 alternate loops for the inner loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
```

12.2 Optimizing Applications Running on Dual-core Processors

Because dual-core systems can run more tasks simultaneously, overall system performance can increase. The trade-offs are that each core has less local memory (because it is shared by the two cores) and less system interconnection bandwidth (which is also shared).

12.2.1 MPI and SHMEM Applications Running under Catamount

By default, processes are placed in rank-sequential order, first on the master core (core 0) on each node and then on the subordinate core (core 1) on each node. So, for a 100-core, 50-node job, the layout of ranks on cores is:

	Node 1	**Node 2**	**Node 3**	**…**	**Node 50**
Core	0 1	0 1	0 1	…	0 1
Rank	0 50	1 51	2 52	…	49 99

Latency times for data transfers between parallel processes can vary according to the type of process-to-core placement: master-to-master, subordinate-to-subordinate, master-to-subordinate on different nodes, and master-to-subordinate on the same node. Master-to master transfers have the shortest latency; subordinate-to-subordinate transfers have the longest latency.

MPI and SHMEM are not aware of the processor placement topology. As a result, some applications may experience performance degradation.

To attain the fastest possible run time, try running your program on the master core of each allocated node. The subordinate cores are allocated to your job but idle.

For example, the command:

```
% yod -sz 64 prog1
```

launches `prog1` on the master core of each of 64 nodes.

The `MPICH_RANK_REORDER_METHOD` environment variable allows you to override the default rank ordering scheme and use an SMP-style placement, a folded-rank placement, or a custom rank placement. See the `intro_mpi`(3) man page for details.

12.2.2 MPI and SHMEM Applications Running under CNL

Processes are placed in packed rank-sequential order, starting with the first node. So, for a 100-core, 50-node job, the layout of ranks on cores is:

	Node 1	Node 2	Node 3	…	Node 50
Core	0 1	0 1	0 1	…	0 1
Rank	0 1	2 3	4 5	…	98 99

Note: You can use the `yod` placement method (rank-sequential order) instead by setting `MPICH_RANK_REORDER_METHOD` to `0`.

To attain the fastest possible run time, try running your program on only one core of each node. (In this case, the other cores are allocated to your job but idle.) This allows each process to have full access to the system interconnection network.

For example, the command:

```
% aprun -n 64 -N 1 ./prog1
```

launches `prog1` on one core of each of 64 dual-core nodes.

Example CNL Applications [13]

This chapter gives examples showing how to compile, link, and run CNL applications.

Verify that your work area is in a Lustre-mounted directory. Then use the `module list` command to verify that the correct modules are loaded. Whenever you compile and link applications to be run under CNL, you need to have the `-cnl` module loaded. Each following example lists the modules that have to be loaded.

Example 3: Basics of running a CNL application

This example shows how to use the PGI C compiler to compile an MPI program and `aprun` to launch the executable.

Modules required:

```
PrgEnv-pgi
xtpe-target-cnl
```

Create a C program, `simple.c`:

```
#include "mpi.h"

int main(int argc, char *argv[])
{
  int rank;
  int numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

  printf("hello from pe %d of %d\n",rank,numprocs);
  MPI_Finalize();
}
```

Compile the program:

```
% cc -o simple simple.c
```

Run the program on six processing elements.

```
% aprun -n 6 ./simple
```

The output to `stdout` will be similar to this:

```
hello from pe 0 of 6
hello from pe 1 of 6
hello from pe 2 of 6
hello from pe 4 of 6
hello from pe 5 of 6
hello from pe 3 of 6
Application 106504 resources: utime 0, stime 0
```

Example 4: Basics of running an MPI application

This example shows how to compile, link, and run an MPI program. The MPI program distributes the work represented in a reduction loop, prints the subtotal for each PE, combines the results from the PEs, and prints the total.

Modules required:

```
PrgEnv-pgi
xtpe-target-cnl
```

Create a Fortran program, `reduce.f90`:

```
program reduce
include "mpif.h"

integer n, nres, ierr

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD,mype,ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD,npes,ierr)

nres = 0
n = 0

do i=mype,100,npes
  n = n + i
enddo

print *, 'My PE:', mype, ' My part:',n

call MPI_REDUCE (n,nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,ierr)

if (mype == 0) print *,'   PE:',mype,'Total is:',nres
```

```
call MPI_FINALIZE (ierr)

end
```

Compile `reduce.f90`:

```
% ftn -o reduce reduce.f90
```

Run the program on two PEs.

```
% aprun -n 2 ./reduce
 My PE:            0  My part:         2550
 My PE:            1  My part:         2500
    PE:            0 Total is:         5050
Application 65539 resources: utime 0, stime 0
```

If desired, you could use this C version of the program:

```
/* program reduce */

#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
  int i, sum, mype, npes, nres, ret;
  ret = MPI_Init (&argc, &argv);
  ret = MPI_Comm_size (MPI_COMM_WORLD, &npes);
  ret = MPI_Comm_rank (MPI_COMM_WORLD, &mype);
  nres = 0;
  sum = 0;
  for (i = mype; i <=100; i += npes) {
    sum = sum + i;
  }

  (void) printf ("My PE:%d  My part:%d\n",mype, sum);
  ret = MPI_Reduce (&sum,&nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  if (mype == 0)
  {
    (void) printf ("PE:%d  Total is:%d\n",mype, nres);
  }
  ret =  MPI_Finalize ();

}
```

Example 5: Running an MPI work distribution program

This example uses MPI solely to identify the processor associated with each process and select the work to be done by each processor. Each processor writes its output directly to `stdout`.

Module required:

```
xtpe-target-cnl
```

Source code of Fortran main program (`prog.f90`):

```
program main
include 'mpif.h'

  call MPI_Init(ierr) ! Required
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  print *,'hello from pe',mype,' of',npes

  do i=1+mype,1000,npes ! Distribute the work
    call work(i,mype)
  enddo

  call MPI_Finalize(ierr) ! Required
end
```

The C function `work.c` processes a single item of work.

Source code of `work.c`:

```
#include <stdio.h>
void work_(int *N, int *MYPE)
{
int n=*N, mype=*MYPE;

  if (n == 42) {
  printf("PE %d: sizeof(long) = %d\n",mype,sizeof(long));
  printf("PE %d: The answer is: %d\n",mype,n);
  }
}
```

Compile `work.c`:

```
% cc -c work.c
```

Compile `prog.f90`, load `work.o`, and create executable `program1`:

```
% ftn -o program1 prog.f90 work.o
```

Run `program1` on two PEs:

```
% aprun -n 2 ./program1
```

Output from `program1`:

```
hello from pe            1  of           2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
 hello from pe            0  of           2
Application 106505 resources: utime 0, stime 0
```

If you want to use a C main program instead of the Fortran main program, compile `prog.c`:

```
#include <stdio.h>
#include <mpi.h>       /* Required */

main(int argc, char **argv)
{
  int i,mype,npes;

  MPI_Init(&argc,&argv);             /* Required */
  MPI_Comm_rank(MPI_COMM_WORLD,&mype);
  MPI_Comm_size(MPI_COMM_WORLD,&npes);

  printf("hello from pe %d of %d\n",mype,npes);

  for (i=1+mype; i<=1000; i+=npes) { /* distribute the work */
    work_(&i, &mype);
  }

  MPI_Finalize();                    /* Required */
}
```

Example 6: Combining results from all processors using MPI

In this example, MPI combines the results from each processor. PE `0` writes the output to `stdout`.

Module required:

```
xtpe-target-cnl
```

Source code of Fortran main program (`prog1.f90`):

```
program main
include 'mpif.h'
integer work1

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  n=0
  do i=1+mype,1000,npes
    n = n + work1(i,mype)
  enddo

  call MPI_Reduce(n,nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,ier)

  if (mype.eq.0) print *,'PE',mype,': The answer is:',nres

  call MPI_Finalize(ierr)
end
```

Source code of `work1.c`:

```
int work1_(int *N, int *MYPE)
{
  int n=*N, mype=*MYPE;
  int mysum=0;

  switch(n) {
    case 12: mysum+=n;
    case 68: mysum+=n;
    case 94: mysum+=n;
    case 120: mysum+=n;
    case 19: mysum-=n;
    case 103: mysum-=n;
```

```
    case 53: mysum-=n;
    case 77: mysum-=n;
  }
  return mysum;
}
```

Compile `work1.c` and `prog1.f90`:

```
% cc -c work1.c
% ftn -o program2 prog1.f90 work1.o
```

To run `program2` on 3 PEs, use:

```
% aprun -n 3 ./program2
 PE            0 : The answer is:        -1184
Application 106506 resources: utime 0, stime 0
```

If you want to use a C main program instead of the Fortran main program, compile `prog1.c`:

```
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  int i,mype,npes,n=0,res;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&mype);
  MPI_Comm_size(MPI_COMM_WORLD,&npes);

  for (i=mype; i<1000; i+=npes) {
    n += work1_(&i, &mype);
  }

  MPI_Reduce(&n,&res,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if (!mype) {
    printf("PE %d: The answer is: %d\n",mype,res);
  }
  MPI_Finalize();
}
```

and link it with `work1.o`:

```
% cc -o program3 prog1.c work1.o
```

Example 7: Using the Cray shmem_put function

This example shows how to use the shmem_put64() function to copy a contiguous data object from the local PE to a contiguous data object on a different PE.

Module required:

```
xtpe-target-cnl
```

Source code of C program (shmem1.c):

```
/*
 *      simple put test
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpp/shmem.h>

/* Dimension of source and target of put operations */
#define DIM     1000000

long target[DIM];
long local[DIM];

main(int argc,char **argv)
{
  register int i;
  int my_partner, my_pe;

  /* Prepare resources required for correct functionality
     of SHMEM on XT3. Alternatively, shmem_init() could
     be called. */
  start_pes(0);

  for (i=0; i<DIM; i++) {
    target[i] = 0L;
    local[i] = shmem_my_pe() + (i * 10);
  }

  my_pe = shmem_my_pe();

  if(shmem_n_pes()%2) {
    if(my_pe == 0) printf("Test needs even number of processes\n");
```

```
    /* Clean up resources before exit. */
    shmem_finalize();
    exit(0);
  }

  shmem_barrier_all();

  /* Test has to be run on two procs. */
  my_partner = my_pe % 2 ? my_pe - 1 : my_pe + 1;

  shmem_put64(target,local,DIM,my_partner);

  /* Synchronize before verifying results. */
  shmem_barrier_all();

  /* Check results of put */
  for(i=0; i<DIM; i++) {
    if(target[i] != (my_partner + (i * 10))) {
      fprintf(stderr,"FAIL (1) on PE %d target[%d] = %d (%d)\n",
        shmem_my_pe(), i, target[i],my_partner+(i*10));
      shmem_finalize();
      exit(-1);
    }
  }

  printf(" PE %d: Test passed.\n",my_pe);

   /* Clean up resources. */
   shmem_finalize();
}
```

Compile shmem1.c and create executable shmem1:

```
% cc -o shmem1 shmem1.c
```

Run shmem1:

```
% aprun -n 4 ./shmem1
 PE 0: Test passed.
 PE 2: Test passed.
 PE 3: Test passed.
 PE 1: Test passed.
Application 106507 resources: utime 0, stime 0
```

Example 8: Using the Cray shmem_get function

This example shows how to use the `shmem_get()` function to copy a contiguous data object from a different PE to a contiguous data object on the local PE.

Module required:

```
xtpe-target-cnl
```

Note: The Fortran module for Cray SHMEM is not supported. Use the `INCLUDE 'mpp/shmem.fh'` statement instead.

Source code of Fortran program (`shmem2.f90`):

```
program reduction
include 'mpp/shmem.fh'

real values, sum
common /c/ values
real work

call start_pes(0)
values=my_pe()
call shmem_barrier_all! Synchronize all PEs
sum = 0.0
do i = 0,num_pes()-1
  call shmem_get(work, values, 1, i)   ! Get next value
  sum = sum + work     ! Sum it
enddo

print*, 'PE',my_pe(),' computedsum=',sum

call shmem_barrier_all
call shmem_finalize

end
```

Compile `shmem2.f90` and create executable `shmem2`:

```
% ftn -o shmem2 shmem2.f90
```

Run shmem2:

```
% aprun -n 2 ./shmem2
 PE             0  computedsum=    1.000000
 PE             1  computedsum=    1.000000
Application 106508 resources: utime 0, stime 0
```

Example 9: Turning off the PGI FORTRAN STOP message

This example shows how to use the NO_STOP_MESSAGE environment variable to turn of the PGI FORTRAN STOP message.

Modules required:

```
xtpe-target-cnl
PrgEnv-pgi
```

Source code of program test_stop.f90:

```
program test_stop

read *, i
  if (i == 1) then
    stop "I was 1"
  else
    stop
  end if
end
```

Compile program test_stop.f90 and create executable test_stop:

```
% ftn -o test_stop test_stop.f90
```

Run test_stop:

```
% aprun -n 2 ./test_stop
1
0
```

Execution results:

```
I was 1
FORTRAN STOP
Application 40962 exit codes: 127
Application 40962 resources: utime 0, stime 0
```

Turn off the FORTRAN STOP messages:

```
% setenv NO_STOP_MESSAGE
```

Run test_stop again:

```
% aprun -n 2 ./test_stop
1
0
```

Execution results:

```
I was 1
Application 40966 exit codes: 127
Application 40966 resources: utime 0, stime 0
```

Example 10: Running an MPI/OpenMP program

This example shows how to compile and run an OpenMP application using PathScale.

Modules required:

```
PrgEnv-pathscale
xtpe-target-cnl
```

Set the OMP_NUM_THREADS environment variable to the number of threads in the team.

Source code of C program omp1.c:

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  int rank, nid, thread;

  MPI_Init(&argc, argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  PMI_CNOS_Get_nid(rank, &nid);
  #pragma omp parallel private(thread)
  {
    thread = omp_get_thread_num();
    #pragma omp barrier
    printf("Hello from rank %d (thread %d) on nid%05d",
```

```
            rank, thread, nid);
    if (thread == 0)
      printf(" <-- master\n");
    else
      printf(" <-- subordinate\n");
  }
  MPI_Finalize();
  return(0);
}
```

Compile and link `omp1.c`:

```
% cc -mp -o omp1 omp1.c
```

Set the OpenMP environment variable:

```
% setenv OMP_NUM_THREADS 2
```

Run program `omp`:

```
% aprun -n 2 -d 2 ./omp1
Hello from rank 0 (thread 0) on nid00540 <-- master
Hello from rank 1 (thread 0) on nid00541 <-- master
Hello from rank 0 (thread 1) on nid00540 <-- subordinate
Hello from rank 1 (thread 1) on nid00541 <-- subordinate
Application 14112 resources: utime 0, stime 0
```

The `aprun` command created two instances of `omp1`; each instance of `omp1` spawned an additional thread.

Example 11: Using a PBS Pro job script

In this example, a PBS Pro job script requests four processors to run an application.

Modules required:

```
xtpe-target-cnl
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `script1`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
```

```
#PBS -q workq
#PBS -l mppwidth=4
# Tell PBS Pro to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid0007/user1
aprun -n 4 ./program1
exit 0
```

Set permissions to executable:

```
% chmod +x script1
```

Submit the job:

```
% qsub script1
```

The `qsub` command produces a batch job log file with output from program1 (see Example 5, page 98). The job log file has the form `script1.o`*nnnnn*.

```
% cat script1.o19850
 hello from pe            0  of           4
 hello from pe            1  of           4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
 hello from pe            3  of           4
 hello from pe            2  of           4
Application 106510 resources: utime 0, stime 0
```

Example 12: Running an MPI program under PBS Pro

This example shows a batch script that runs the program `simple.c` (see Example 3, page 95).

Modules required:

```
xtpe-target-cnl
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `script2`:

```
% cat script2
#PBS -l mppwidth=6
#PBS -joe
cd /lus/nid00011/user1
aprun -n 6 ./simple
```

Set permissions to executable:

```
% chmod +x script2
```

Submit the script to the PBS Pro batch system:

```
% qsub script2
```

Display the job results:

```
% cat script2.o19852
hello from pe 0 of 6
hello from pe 2 of 6
hello from pe 3 of 6
hello from pe 1 of 6
hello from pe 4 of 6
hello from pe 5 of 6
Application 106513 resources: utime 0, stime 0
```

Example 13: Running an MPI_REDUCE program under PBS Pro

This example shows a batch script that runs the program `reduce.f90` (see Example 4, page 96).

Modules required:

```
xtpe-target-cnl
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create a batch script, `run_reduce`, verifying that the executable is in a directory in the Lustre file system:

```
#!/bin/sh
#PBS -l mppwidth=2
#PBS -joe
#PBS -l walltime=00:30:00
cd $HOME/pe_user/
echo "Running the Example reduce "
```

```
echo ""
date
echo ""
cd /lus/nid00011/user1
aprun -n 2 ./reduce
```

Set permissions to executable:

```
% chmod +x run_reduce
```

Submit the script to the PBS Pro batch system:

```
% qsub run_reduce
```

Display the job results:

```
% cat run_reduce.o70977
Running the Example reduce

Wed May  9 13:36:52 CDT 2007

 My PE:            1  My part:         2500
 My PE:            0  My part:         2550
    PE:            0 Total is:         5050
Application 65545 resources: utime 0, stime 0
```

Example 14: Using a script to create and run a batch job

This example script takes two arguments, the name of a program (`shmem2`, see Example 8, page 104) and the number of processors on which to run the program. The script performs the following actions:

1. Creates a temporary file that contains a PBS Pro batch job script
2. Submits the file to PBS Pro
3. Deletes the temporary file

Modules required:

```
xtpe-target-cnl
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `run123`:

```
#!/bin/csh
```

```
  if ( "$1" == "" ) then
    echo "Usage: run [executable|script] [ncpus]"
    exit
  endif
  set n=1 # set default number of CPUs
  if ( "$2" != "" ) set n=$2
  cat > job.$$ <<EOT #creates the batch jobscript
#!/bin/csh
#PBS -N $1
#PBS -l mppwidth=$n
#PBS -joe
cd ${PWD}
aprun -n $n -t30 ./$1
EOT
qsub job.$$ # submit batch job
rm job.$$
```

Set file permissions to executable:

```
% chmod +x run123
```

Run the job script:

```
% ./run123 shmem2 2
```

List the job output:

```
% cat shmem2.o73595
 PE             0  computedsum=    1.000000
 PE             1  computedsum=    1.000000
Application 35612 resources: utime 0, stime 0
```

Example 15: Running multiple sequential applications

To run multiple sequential applications, the number of processors you specify as an argument to `qsub` must be equal to or greater than the **largest number** of processors required by a single invocation of `aprun` in your script. For example, in job script `mult_seq_cnl`, the `-l mppwidth` value is 4 because the largest `aprun n` value is 4.

Modules required:

```
xtpe-target-cnl
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `mult_seq_cnl`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=4
# Tell PBS Pro to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid00011/user1
aprun -n 2 ./program1
aprun -n 3 ./program2
aprun -n 4 ./shmem1
aprun -n 2 ./shmem2
exit 0
```

The script launches applications `program1` (see Example 5, page 98), `program2` (see Example 6, page 100), `shmem1` (see Example 7, page 102), and `shmem2` (see Example 8, page 104).

Set file permission to executable:

```
% chmod +x mult_seq_cnl
```

Run the script:

```
% qsub mult_seq_cnl
```

List the output:

```
% cat mult_seq_cnl.o19884
 hello from pe            1  of           2
 hello from pe            0  of           2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Application 106691 resources: utime 0, stime 0
PE           0 : The answer is:        -1184
Application 106692 resources: utime 0, stime 0
 PE 0: Test passed.
 PE 3: Test passed.
 PE 2: Test passed.
 PE 1: Test passed.
Application 106693 resources: utime 0, stime 0
 PE            0  computedsum=    1.000000
```

```
 PE             1  computedsum=    1.000000
Application 106694 resources: utime 0, stime 0
```

Example 16: Running multiple parallel applications

If you are running multiple parallel applications, the number of processors must be equal to or greater than the **total** number of processors specified by calls to `aprun`. For example, in job script `mult_par_cnl`, the `-l mppwidth` value is `11` because the total of the `aprun n` values is `11`.

Modules required:

```
xtpe-target-cnl
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `mult_par_cnl`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=11
# Tell PBS Pro to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid00011/user1
aprun -n 2 ./program1 &
aprun -n 3 ./program2 &
aprun -n 4 ./shmem1 &
aprun -n 2 ./shmem2 &
exit 0
```

The script launches applications `program1` (see Example 5, page 98), `program2` (see Example 6, page 100), `shmem1` (see Example 7, page 102), and `shmem2` (see Example 8, page 104).

Set file permission to executable:

```
% chmod +x mult_par_cnl
```

Run the script:

```
% qsub mult_par_cnl
```

List the output:

```
% cat mult_par_cnl.o7231
 hello from pe            0  of            2
 hello from pe            1  of            2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Application 155001 resources: utime 0, stime 0
 PE            0 : The answer is:        -1184
Application 155002 resources: utime 0, stime 0
 PE 0: Test passed.
 PE 3: Test passed.
 PE 2: Test passed.
 PE 1: Test passed.
Application 155003 resources: utime 0, stime 0
 PE             0  computedsum=    1.000000
 PE             1  computedsum=    1.000000
Application 155004 resources: utime 0, stime 0
```

Example 17: Using the high-level PAPI interface

PAPI provides simple high-level interfaces for instrumenting applications written in C or Fortran. This example shows the use of the `PAPI_start_counters()` and `PAPI_stop_counters()` functions.

Modules required:

```
xtpe-target-cnl
papi-cnl
```

Source of `papi_hl.c`:

```
#include <papi.h>
void main()
{

  int retval, Events[2]= {PAPI_TOT_CYC, PAPI_TOT_INS};
  long_long values[2];

  if (PAPI_start_counters (Events, 2) != PAPI_OK) {
    printf("Error starting counters\n");
    exit(1);
  }

  /* Do some computation here... */
```

```
  if (PAPI_stop_counters (values, 2) != PAPI_OK) {
    printf("Error stopping counters\n");
    exit(1);
  }

  printf("PAPI_TOT_CYC = %lld\n", values[0]);
  printf("PAPI_TOT_INS = %lld\n", values[1]);
}
```

Compile `papi_hl.c`:

```
% cc -o papi_hl papi_hl.c
```

Run `papi_hl`:

```
% aprun ./papi_hl
PAPI_TOT_CYC = 3350
PAPI_TOT_INS = 215
Application 155005 exit codes: 19
Application 155005 resources: utime 0, stime 0
```

Example 18: Using the low-level PAPI interface

PAPI provides an advanced low-level interface for instrumenting applications. The PAPI library must be initialized before calling any of these functions; initialization can be done by issuing either a high-level function call or a call to `PAPI_library_init()`. This example shows the use of the `PAPI_create_eventset()`, `PAPI_add_event()`, `PAPI_start()`, and `PAPI_read()` functions.

Modules required:

```
xtpe-target-cnl
papi-cnl
```

Source of `papi_ll.c`:

```
#include <papi.h>
void main()
{
  int EventSet = PAPI_NULL;
  long_long values[1];

  /* Initialize PAPI library */
  if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
```

```
    printf("Error initializing PAPI library\n");
    exit(1);
  }

  /* Create Event Set */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK) {
    printf("Error creating eventset\n");
  exit(1);
  }

  /* Add Total Instructions Executed to eventset */
  if (PAPI_add_event (EventSet, PAPI_TOT_INS) != PAPI_OK) {
    printf("Error adding event\n");
    exit(1);
  }

  /* Start counting ... */
  if (PAPI_start (EventSet) != PAPI_OK) {
    printf("Error starting counts\n");
    exit(1);
  }

  /* Do some computation here...*/

  if (PAPI_read (EventSet, values) != PAPI_OK) {
    printf("Error stopping counts\n");
    exit(1);
  }
  printf("PAPI_TOT_INS = %lld\n", values[0]);
}
```

Compile `papi_ll.c`:

```
% cc -o papi_ll papi_ll.c
```

Run `papi_ll`:

```
% aprun ./papi_ll
PAPI_TOT_INS = 103
Application 155006 exit codes: 19
Application 155006 resources: utime 0, stime 0
```

Example 19: Using basic CrayPat functions

This example shows how to instrument a program, run the instrumented program, and generate CrayPat reports.

Modules required:

```
xtpe-target-cnl
craypat
```

Compile the sample program `prog.f90` and the routine it calls, `work.c`.

Source code of `prog.f90`:

```
program main
include 'mpif.h'

  call MPI_Init(ierr)     ! Required
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  print *,'hello from pe',mype,' of',npes

  do i=1+mype,1000,npes   ! Distribute the work
    call work(i,mype)
  enddo

  call MPI_Finalize(ierr) ! Required
end
```

Source code of `work.c`:

```
void work_(int *N, int *MYPE)
{
  int n=*N, mype=*MYPE;

  if (n == 42) {
    printf("PE %d: sizeof(long) = %d\n",mype,sizeof(long));
    printf("PE %d: The answer is: %d\n",mype,n);
  }
}
```

Compile `prog.f90` and `work.c` and create executable `program1`:

```
% cc -c work.c
% ftn -o program1  prog.f90 work.o
```

Run pat_build to generate instrumented program program1+pat:

```
% pat_build -u -g mpi program1 program1+pat
INFO: A trace intercept routine was created for the function 'work_'.
INFO: a total of 39 function entry points were traced
```

The tracegroup (-g option) is mpi.

Run program1+pat:

```
% aprun -n 4 ./program1+pat
 hello from pe            1  of            4
 hello from pe            3  of            4
 hello from pe            2  of            4
 hello from pe            0  of            4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Experiment data directory written:
/ufs/home/users/user1/pat/program1+pat+3820tdt
```

Note: When executed, the instrumented executable creates directory *progname*+pat+*PIDkeyletters*, where . *PID* is the process ID that was assigned to the instrumented program at run time.

Run pat_report to generate reports program1.rpt1 (using default pat_report options) and program1.rpt2 (using the -O calltree option).

```
% pat_report program1+pat+3820tdt  > program1.rpt1
Data file 4/4: [.....................]
% pat_report -O calltree program1+pat+3820tdt > program1.rpt2
Data file 4/4: [.....................]
```

List program1.rpt1:

```
% more program1.rpt1
CrayPat/X:  Version 3.2 Revision 799 (xf 784)  04/23/07 07:49:22

Experiment:  trace

Experiment data file:
  /lus/nid00011/user1/cnl/program1+pat+3820tdt/*.xf  (RTS)

Original program:  /lus/nid00011/user1/cnl/program1

Instrumented with:  pat_build -u -g mpi program1 program1+pat
```

```
Instrumented program:  /lus/nid00011/user1/cnl/./program1+pat

Program invocation:  ./program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/2.0.05/mpich2-64
  MPICH_DIR=/opt/xt-mpt/2.0.05/mpich2-64/P2
  MPICH_DIR_FTN_DEFAULT64=/opt/xt-mpt/2.0.05/mpich2-64/P2W
  PAT_BUILD_ASYNC=0
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx
  PAT_RT_EXPFILE_PER_PROCESS=1
  PAT_RT_HWPC=1

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx

Report command line options:  <none>

System type and speed:   x86_64  2400 MHz

Operating system:
  Linux 2.6.16.27-0.9-cnl #1 SMP Tue May 8 18:24:11 PDT 2007

Hardware performance counter events:
  PAPI_TLB_DM        Data translation lookaside buffer misses
  PAPI_L1_DCA        Level 1 data cache accesses
  PAPI_FP_OPS        Floating point operations
  DATA_CACHE_MISSES  Data Cache Misses
  User_Cycles        Virtual Cycles

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    PAPI_TLB_DM            0.000  misses
    PAPI_L1_DCA         1282.080  ops
    PAPI_FP_OPS            3.000  ops
    DATA_CACHE_MISSES      8.312  misses
    User_Cycles         4302.000  cycles
    Time                   1.799  microseconds
```

```
Number of traced functions: 42

Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
    -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b ex,gr,fu,pe=HIDE,th=HIDE

  Options for related tables not shown by default:
    -O callers
    -O callers+src
    -O calltree
    -O calltree+src

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).

Table 1:  Profile by Function Group and Function

Experiment=1 / Group / Function / PE='HIDE' / Thread=0='HIDE'

==============================================================================
Totals for program
------------------------------------------------------------------------------
  Time%                                    100.0%
  Time                                  0.001362
  Imb.Time                                    --
  Imb.Time%                                   --
  Calls                                     2628
  PAPI_TLB_DM              0.712M/sec        881 misses
  PAPI_L1_DCA           1173.861M/sec    1452993 ops
  PAPI_FP_OPS              5.548M/sec       6867 ops
  DATA_CACHE_MISSES       11.104M/sec      13745 misses
  User time                0.001 secs    2970696 cycles
  Utilization rate                         90.9%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time    5.548M/sec       6867 ops        0.0%peak
  HW FP Ops / WCT          5.043M/sec
```

```
  Computation intensity                    0.00 ops/ref
  LD & ST per TLB miss                  1649.25 refs/miss
  LD & ST per D1 miss                    105.71 refs/miss
  D1 cache hit ratio                      99.1%
  % TLB misses / cycle                     0.0%
=======================================================================
       88.2%
  HW FP Ops / Cycles                       0.00 ops/cycle
  HW FP Ops / User time     4.585M/sec      4331 ops        0.0%peak
  HW FP Ops / WCT           4.042M/sec
  Computation intensity                    0.00 ops/ref
  LD & ST per TLB miss                  1147.43 refs/miss
  LD & ST per D1 miss                    114.77 refs/miss
  D1 cache hit ratio                      99.1%
  % TLB misses / cycle                     0.0%
=======================================================================

<snip>

Notes for table 3:

  Table option:
    -O program_time
  Options implied by table option:
    -d pt -b ex,pe,th=[mmm]

Table 3:  Program Wall Clock Time

  Process |Experiment=1
     Time |PE
          | Thread=0[mmm]

 0.008343 |Total
|------------------------
| 0.009220 |pe.1
| 0.009074 |pe.0
| 0.007577 |pe.2
| 0.007501 |pe.3
|========================
```

List `program1.rpt2`:

```
% more program1.rpt2
CrayPat/X:  Version 3.2 Revision 799 (xf 784)  04/23/07 07:49:22
```

```
Experiment:  trace

Experiment data file:
  /lus/nid00011/user1/cnl/program1+pat+3820tdt/*.xf  (RTS)

Original program:  /lus/nid00011/user1/cnl/program1

Instrumented with:  pat_build -u -g mpi program1 program1+pat

Instrumented program:  /lus/nid00011/user1/cnl/./program1+pat

Program invocation:  ./program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/2.0.05/mpich2-64
  MPICH_DIR=/opt/xt-mpt/2.0.05/mpich2-64/P2
  MPICH_DIR_FTN_DEFAULT64=/opt/xt-mpt/2.0.05/mpich2-64/P2W
  PAT_BUILD_ASYNC=0
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx
  PAT_RT_EXPFILE_PER_PROCESS=1
  PAT_RT_HWPC=1

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx

Report command line options:  -O calltree

System type and speed:   x86_64  2400 MHz

Operating system:
  Linux 2.6.16.27-0.9-cnl #1 SMP Tue May 8 18:24:11 PDT 2007

Hardware performance counter events:
  PAPI_TLB_DM        Data translation lookaside buffer misses
  PAPI_L1_DCA        Level 1 data cache accesses
  PAPI_FP_OPS        Floating point operations
  DATA_CACHE_MISSES  Data Cache Misses
  User_Cycles        Virtual Cycles
```

```
Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    PAPI_TLB_DM            0.000  misses
    PAPI_L1_DCA         1282.080  ops
    PAPI_FP_OPS            3.000  ops
    DATA_CACHE_MISSES      8.312  misses
    User_Cycles         4302.000  cycles
    Time                   1.799  microseconds

Number of traced functions: 42

Notes for table 1:

  Table option:
    -O calltree
  Options implied by table option:
    -d ti%@0.05,cum_ti%,ti,tr,P -b ex,ct,pe=HIDE,th=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).

Table 1:  Function Calltree View

Experiment=1 / Calltree / PE='HIDE' / Thread=0='HIDE'

==============================================================================
Totals for program
------------------------------------------------------------------------------
  Time%                                    100.0%
  Cum.Time%                                100.0%
  Time                                  0.001362
  Calls                                     2628
  PAPI_TLB_DM              0.712M/sec        881 misses
  PAPI_L1_DCA           1173.861M/sec    1452993 ops
  PAPI_FP_OPS              5.548M/sec       6867 ops
  DATA_CACHE_MISSES       11.104M/sec      13745 misses
  User time                0.001 secs    2970696 cycles
  Utilization rate                         90.9%
  HW FP Ops / Cycles                        0.00 ops/cycle
```

```
  HW FP Ops / User time     5.548M/sec      6867 ops        0.0%peak
  HW FP Ops / User time     5.548M/sec      6867 ops        0.0%peak
  HW FP Ops / WCT           5.043M/sec
  Computation intensity                      0.00 ops/ref
  LD & ST per TLB miss                    1649.25 refs/miss
  LD & ST per D1 miss                      105.71 refs/miss
  D1 cache hit ratio                        99.1%
  % TLB misses / cycle                       0.0%
==========================================================================

<snip>

exit
--------------------------------------------------------------------------
  Time%                                     10.0%
  Cum.Time%                                100.0%
  Time                                  0.000136
  Calls                                      800
  PAPI_TLB_DM                                  0 misses
  PAPI_L1_DCA            1735.094M/sec     236515 ops
  PAPI_FP_OPS               9.243M/sec       1260 ops
  DATA_CACHE_MISSES        14.005M/sec       1909 misses
  User time                 0.000 secs      327150 cycles
  Utilization rate                         100.0%
  HW FP Ops / Cycles                         0.00 ops/cycle
  HW FP Ops / User time     9.243M/sec       1260 ops        0.0%peak
  HW FP Ops / WCT           9.243M/sec
  Computation intensity                      0.01 ops/ref
  LD & ST per TLB miss                  236515.00 refs/miss
  LD & ST per D1 miss                      123.89 refs/miss
  D1 cache hit ratio                        99.2%
  % TLB misses / cycle                       0.0%
==========================================================================
```

Example 20: Using hardware performance counters

This example uses the same instrumented program as Example 19, page 117 and generates reports showing hardware performance counter (HWPC) information.

Modules required:

```
xtpe-target-cnl
craypat
```

Collect HWPC event set 1 information and generate report `program1.rpt3` (for a list of predefined event sets, see the hwpc(3) man page):

```
% setenv PAT_RT_HWPC 1
% aprun -n 4 ./program1+pat
CrayPat/X:  Version 3.1 Revision 363  08/28/06 16:25:58
 hello from pe            3  of            4
 hello from pe            1  of            4
 hello from pe            2  of            4
 hello from pe            0  of            4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Experiment data directory written:
/ufs/home/users/user1/pat/program1+pat+3820tdt
% pat_report program1+pat+3820tdt > program1.rpt3
Data file 4/4:
[....................]
```

List `program1.rpt3`:

```
Experiment:  trace

Experiment data file:
  /lus/nid00011/user1/cnl/program1+pat+3820tdt/*.xf  (RTS)

Original program:  /lus/nid00011/user1/cnl/program1

Instrumented with:  pat_build -u -g mpi program1 program1+pat

Instrumented program:  /lus/nid00011/user1/cnl/./program1+pat

Program invocation:  ./program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/2.0.05/mpich2-64
  MPICH_DIR=/opt/xt-mpt/2.0.05/mpich2-64/P2
  MPICH_DIR_FTN_DEFAULT64=/opt/xt-mpt/2.0.05/mpich2-64/P2W
  PAT_BUILD_ASYNC=0
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx
  PAT_RT_EXPFILE_PER_PROCESS=1
```

```
  PAT_RT_HWPC=1

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx

Report command line options:  <none>

System type and speed:   x86_64  2400 MHz

Operating system:
  Linux 2.6.16.27-0.9-cnl #1 SMP Tue May 8 18:24:11 PDT 2007

Hardware performance counter events:
  PAPI_TLB_DM        Data translation lookaside buffer misses
  PAPI_L1_DCA        Level 1 data cache accesses
  PAPI_FP_OPS        Floating point operations
  DATA_CACHE_MISSES  Data Cache Misses
  User_Cycles        Virtual Cycles

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    PAPI_TLB_DM            0.000  misses
    PAPI_L1_DCA         1282.080  ops
    PAPI_FP_OPS            3.000  ops
    DATA_CACHE_MISSES      8.312  misses
    User_Cycles         4302.000  cycles
    Time                   1.799  microseconds

Number of traced functions: 42

Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
    -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P \
    -b ex,gr,fu,pe=HIDE,th=HIDE

  Options for related tables not shown by default:
    -O load_balance
    -O callers
    -O callers+src
```

```
    -O calltree
    -O calltree+src

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).

Table 1:  Profile by Function Group and Function

Experiment=1 / Group / Function / PE='HIDE' / Thread=0='HIDE'

======================================================================
Totals for program
----------------------------------------------------------------------
  Time%                                  100.0%
  Time                                0.001362
  Imb.Time                                --
  Imb.Time%                               --
  Calls                                 2628
  PAPI_TLB_DM              0.712M/sec    881 misses
  PAPI_L1_DCA           1173.861M/sec 1452993 ops
  PAPI_FP_OPS              5.548M/sec    6867 ops
  DATA_CACHE_MISSES       11.104M/sec   13745 misses
  User time                0.001 secs 2970696 cycles
  Utilization rate                      90.9%
  HW FP Ops / Cycles                     0.00 ops/cycle
  HW FP Ops / User time    5.548M/sec    6867 ops        0.0%peak
  HW FP Ops / WCT          5.043M/sec
  Computation intensity                  0.00 ops/ref
  LD & ST per TLB miss                1649.25 refs/miss
  LD & ST per D1 miss                  105.71 refs/miss
  D1 cache hit ratio                    99.1%
  % TLB misses / cycle                   0.0%
======================================================================
<snip>

Notes for table 3:

  Table option:
    -O program_time
  Options implied by table option:
    -d pt -b ex,pe,th=[mmm]
```

```
Table 3:  Program Wall Clock Time

  Process |Experiment=1
     Time |PE
          | Thread=0[mmm]

 0.008343 |Total
|-----------------------
| 0.009220 |pe.1
| 0.009074 |pe.0
| 0.007577 |pe.2
| 0.007501 |pe.3
|=======================
```

Collect information about translation lookaside buffer (TLB) misses (PAPI_TLB_DM) and generate report program1.rpt4:

```
% setenv PAT_RT_HWPC PAPI_TLB_DM
% aprun -n 4 ./program1+pat
 hello from pe 0  of 4
 hello from pe 1  of 4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
 hello from pe 2  of 4
 hello from pe 3  of 4
Experiment data file written:
/lus/nid00011/user1/cnl/program1+pat+3820tdt
Application 34876 resources: utime 0, stime 0
% pat_report program1+pat+2790tdt.xf > program1.rpt4
Data file 4/4: [....................]
```

List program1.rpt4:

```
CrayPat/X:  Version 3.2 Revision 799 (xf 784)  04/23/07 07:49:22

Experiment:  trace

Experiment data file:
  /lus/nid00011/user1/cnl/program1+pat+2790tdt.xf  (RTS)

Original program:  /lus/nid00011/user1/cnl/program1

Instrumented with:  pat_build -u -g mpi program1 program1+pat
```

```
Instrumented program:  /lus/nid00011/user1/cnl/./program1+pat

Program invocation:  ./program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/2.0.05/mpich2-64
  MPICH_DIR=/opt/xt-mpt/2.0.05/mpich2-64/P2
  MPICH_DIR_FTN_DEFAULT64=/opt/xt-mpt/2.0.05/mpich2-64/P2W
  PAT_RT_HWPC=PAPI_TLB_DM

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx

Report command line options:  <none>

System type and speed:   x86_64  2400 MHz

Operating system:
  Linux 2.6.16.27-0.9-cnl #1 SMP Tue May 8 18:24:11 PDT 2007

Hardware performance counter events:
  PAPI_TLB_DM  Data translation lookaside buffer misses
  User_Cycles  Virtual Cycles

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    PAPI_TLB_DM      0.000  misses
    User_Cycles   3690.000  cycles
    Time             1.546  microseconds

Number of traced functions: 42

Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
```

```
    -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P -b gr,fu,pe=HIDE,th=HIDE

  Options for related tables not shown by default:
    -O load_balance
    -O callers
    -O callers+src
    -O calltree
    -O calltree+src

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).

Table 1:  Profile by Function Group and Function

Group / Function / PE='HIDE' / Thread=0='HIDE'

==============================================================================
Totals for program
------------------------------------------------------------------------------
  Time%                                 100.0%
  Time                                0.001136
  Imb.Time                                --
  Imb.Time%                               --
  Calls                                 2628
  PAPI_TLB_DM           0.788M/sec       833 misses
  User time             0.001 secs   2538210 cycles
  Utilization rate                     93.1%
  % TLB misses / cycle                  0.0%
==============================================================================
<snip>
Notes for table 3:

  Table option:
    -O program_time
  Options implied by table option:
    -d pt -b pe,th=[mmm]

Table 3:  Program Wall Clock Time

  Process |PE
     Time | Thread=0[mmm]
```

```
 0.132561 |Total
|------------------------
| 0.140586 |pe.3
| 0.140554 |pe.2
| 0.124558 |pe.1
| 0.124545 |pe.0
|========================
```

Example Catamount Applications [14]

This chapter gives examples showing how to compile, link, and run Catamount applications. Use the `module list` command to verify that the correct modules are loaded. If the `xtpe-target-cnl` module is loaded, use:

```
% module swap xtpe-target-cnl xtpe-target-catamount
```

Each following example lists the additional modules that have to be loaded.

Example 21: Basics of running a Catamount application

This example shows how to use the PGI C compiler to compile an MPI program and `yod` to launch the executable.

Modules required:

```
xtpe-target-catamount
PrgEnv-pgi
```

Create a C program, `simple.c`:

```
#include "mpi.h"

int main(int argc, char *argv[])
{
  int rank;
  int numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

  printf("hello from pe %d of %d\n",rank,numprocs);
  MPI_Finalize();
}
```

Compile the program:

```
% cc -o simple simple.c
```

Run the program:

```
% yod -sz 6 simple
hello from pe 3 of 6
hello from pe 0 of 6
hello from pe 3 of 6
```

```
hello from pe 5 of 6
hello from pe 2 of 6
hello from pe 1 of 6
hello from pe 4 of 6
```

Example 22: Basics of running an MPI application

This example shows how to compile, link, and run an MPI program. The MPI program distributes the work represented in a reduction loop, prints the subtotal for each PE, combines the results from the PEs, and prints the total.

Module required:

```
xtpe-target-catamount
```

Create a Fortran program, `reduce.f90`:

```
program reduce
include "mpif.h"

integer n, nres, ierr

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD,mype,ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD,npes,ierr)

nres = 0
n = 0

do i=mype,100,npes
  n = n + i
enddo

print *, 'My PE:', mype, ' My part:',n

call MPI_REDUCE (n,nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,ierr)

if (mype == 0) print *,'   PE:',mype,'Total is:',nres

call MPI_FINALIZE (ierr)

end
```

Compile `reduce.f90` and create executable `reduce`:

```
% ftn -o reduce reduce.f90
```

Run the program:

```
% yod -sz 2  reduce
 My PE:            0  My part:         2550
 My PE:            1  My part:         2500
    PE:            0 Total is:         5050
```

If desired, you could use this C version of the program:

```
/* program reduce */

#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
  int i, sum, mype, npes, nres, ret;
  ret = MPI_Init (&argc, &argv);
  ret = MPI_Comm_size (MPI_COMM_WORLD, &npes);
  ret = MPI_Comm_rank (MPI_COMM_WORLD, &mype);
  nres = 0;
  sum = 0;
  for (i = mype; i <=100; i += npes) {
    sum = sum + i;
  }

  (void) printf ("My PE:%d  My part:%d\n",mype, sum);
  ret = MPI_Reduce (&sum,&nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  if (mype == 0)
  {
    (void) printf ("PE:%d  Total is:%d\n",mype, nres);
  }
  ret =  MPI_Finalize ();

}
```

Example 23: Running an MPI work distribution program

This example uses MPI solely to identify the processor associated with each process and select the work to be done by each processor. Each processor writes its output directly to `stdout`.

Module required:

```
xtpe-target-catamount
```

Source code of Fortran main program (`prog.f90`):

```
program main
include 'mpif.h'

  call MPI_Init(ierr) ! Required
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  print *,'hello from pe',mype,' of',npes

  do i=1+mype,1000,npes ! Distribute the work
    call work(i,mype)
  enddo

  call MPI_Finalize(ierr) ! Required
end
```

The C function `work.c` processes a single item of work.

Source code of `work.c`:

```
#include <stdio.h>
void work_(int *N, int *MYPE)
{
int n=*N, mype=*MYPE;

  if (n == 42) {
  printf("PE %d: sizeof(long) = %d\n",mype,sizeof(long));
  printf("PE %d: The answer is: %d\n",mype,n);
  }
}
```

Compile `work.c`:

```
% cc -c work.c
```

Compile `prog.f90`, load `work.o`, and create executable `program1`:

```
% ftn -o program1 prog.f90 work.o
```

Run `program1`:

```
% yod -sz 2 program1
```

Output from `program1`:

```
hello from pe            0  of           2
hello from pe            1  of           2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
```

If you want to use a C main program instead of the Fortran main program, compile `prog.c`:

```
#include <stdio.h>
#include <mpi.h>       /* Required */

main(int argc, char **argv)
{
  int i,mype,npes;

  MPI_Init(&argc,&argv);             /* Required */
  MPI_Comm_rank(MPI_COMM_WORLD,&mype);
  MPI_Comm_size(MPI_COMM_WORLD,&npes);

  printf("hello from pe %d of %d\n",mype,npes);

  for (i=1+mype; i<=1000; i+=npes) { /* distribute the work */
    work_(&i, &mype);
  }

  MPI_Finalize();                    /* Required */
}
```

Example 24: Combining results from all processors using MPI

In this example, MPI combines the results from each processor. PE `0` writes the output to `stdout`.

Module required:

```
xtpe-target-catamount
```

Source code of Fortran main program (`prog1.f90`):

```
program main
include 'mpif.h'
integer work1

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  n=0
  do i=1+mype,1000,npes
    n = n + work1(i,mype)
  enddo

  call MPI_Reduce(n,nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,ier)

  if (mype.eq.0) print *,'PE',mype,': The answer is:',nres

  call MPI_Finalize(ierr)
end
```

The C function `work1.c` processes a single item of work.

Source code of `work1.c`:

```
int work1_(int *N, int *MYPE)
{
  int n=*N, mype=*MYPE;
  int mysum=0;

  switch(n) {
    case 12: mysum+=n;
    case 68: mysum+=n;
    case 94: mysum+=n;
    case 120: mysum+=n;
    case 19: mysum-=n;
    case 103: mysum-=n;
    case 53: mysum-=n;
    case 77: mysum-=n;
  }
  return mysum;
}
```

Compile work1.c and prog1.f90:

```
% cc -c work1.c
% ftn -o program2 prog1.f90 work1.o
```

Run program2:

```
% yod -sz 3 program2
PE            0 : The answer is:         -1184
```

If you want to use a C main program instead of the Fortran main program, compile prog1.c:

```
#include <stdio.h>
#include <mpi.h>

main(int argc, char **argv)
{
  int i,mype,npes,n=0,res;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&mype);
  MPI_Comm_size(MPI_COMM_WORLD,&npes);

  for (i=mype; i<1000; i+=npes) {
    n += work1_(&i, &mype);
  }

  MPI_Reduce(&n,&res,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if (!mype) {
    printf("PE %d: The answer is: %d\n",mype,res);
  }
  MPI_Finalize();
}
```

and link it with work1.o:

```
% cc -o program3 prog1.c work1.o
```

Example 25: Using the Cray shmem_put function

This example shows how to use the shmem_put64() function to copy a contiguous data object from the local PE to a contiguous data object on a different PE.

Module required:

```
xtpe-target-catamount
```

Source code of C program (shmem1.c):

```
/*
 *      simple put test
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpp/shmem.h>

/* Dimension of source and target of put operations */
#define DIM 1000000

long target[DIM];
long local[DIM];

main(int argc,char **argv)
{
  register int i;
  int my_partner, my_pe;

  /* Prepare resources required for correct functionality
     of SHMEM on XT3. Alternatively, shmem_init() could
     be called. */
  start_pes(0);

  for (i=0; i<DIM; i++) {
    target[i] = 0L;
    local[i] = shmem_my_pe() + (i * 10);
  }

  my_pe = shmem_my_pe();

  if(shmem_n_pes()%2) {
    if(my_pe == 0) printf("Test needs even number of processes\n");
    /* Clean up resources before exit. */
    shmem_finalize();
    exit(0);
  }
```

```
  shmem_barrier_all();

  /* Test has to be run on two procs. */
  my_partner = my_pe % 2 ? my_pe - 1 : my_pe + 1;

  shmem_put64(target,local,DIM,my_partner);

  /* Synchronize before verifying results. */
  shmem_barrier_all();

  /* Check results of put */
  for(i=0; i<DIM; i++) {
    if(target[i] != (my_partner + (i * 10))) {
      fprintf(stderr,"FAIL (1) on PE %d target[%d] = %d (%d)\n",
        shmem_my_pe(), i, target[i],my_partner+(i*10));
      shmem_finalize();
      exit(-1);
    }
  }

  printf(" PE %d: Test passed.\n",my_pe);

   /* Clean up resources. */
   shmem_finalize();
}
```

Compile `shmem1.c` and create executable `shmem1`:

```
% cc -o shmem1 shmem1.c
```

Run `shmem1`:

```
% yod -sz 4 shmem1
 PE 2: Test passed.
 PE 1: Test passed.
 PE 3: Test passed.
 PE 0: Test passed.
```

Example 26: Using the Cray shmem_get function

This example shows how to use the `shmem_get` function to copy a contiguous data object from a different PE to a contiguous data object on the local PE.

Note: The Fortran module for Cray SHMEM is not supported. Use the `INCLUDE 'mpp/shmem.fh'` statement instead.

Module required:

```
xtpe-target-catamount
```

Source code of Fortran program (shmem2.f90):

```
program reduction
include 'mpp/shmem.fh'

real values, sum
common /c/ values
real work

call start_pes(0)
values=my_pe()
call shmem_barrier_all! Synchronize all PEs
sum = 0.0
do i = 0,num_pes()-1
  call shmem_get(work, values, 1, i)   ! Get next value
  sum = sum + work     ! Sum it
enddo

print*, 'PE',my_pe(),' computedsum=',sum

call shmem_barrier_all
call shmem_finalize

end
```

Compile shmem2.f90 and create executable shmem2:

```
% ftn -o shmem2 shmem2.f90
```

Run shmem2:

```
% yod -np 2 shmem2
PE             0  computedsum=    1.000000
PE             1  computedsum=    1.000000
```

Example 27: Turning off the PGI FORTRAN STOP message

This example shows how to use the NO_STOP_MESSAGE environment variable to turn of the FORTRAN STOP message.

Modules required:

```
xtpe-target-catamount
```

```
PrgEnv-pgi
```

Source code of program `test_stop.f90`:

```
program test_stop

read *, i
  if (i == 1) then
    stop "I was 1"
  else
    stop
  end if
end
```

Verify that the `PrgEnv-pgi` module is loaded.

Compile program `test_stop.f90` and create executable `test_stop`:

```
% ftn -o test_stop test_stop.f90
```

Run `test_stop`:

```
% yod -sz 2 test_stop
1
0
```

Execution results:

```
I was 1
FORTRAN STOP
```

Turn off the `FORTRAN STOP` messages:

```
% setenv NO_STOP_MESSAGE
```

Run `test_stop again`:

```
% yod -sz 2 test_stop
1
0
```

Execution results:

```
I was 1
```

Example 28: Using `dclock()` to calculate elapsed time

The following example uses the `dclock()` function to calculate the elapsed time of a program segment.

Module required:

```
xtpe-target-catamount
```

Source code of `dclock.c`:

```
#include <catamount/dclock.h>

main()
{
  double start_time, end_time, elapsed_time;
  start_time = dclock();
  sleep(5);
  end_time = dclock();
  elapsed_time = end_time - start_time;
  printf("\nElapsed time = %f\n",elapsed_time);
}
```

Compile `dclock.c` and create executable `dclock`:

```
% cc -o dclock dclock.c
```

Run `dclock`:

```
% yod dclock
Elapsed time = 5.000007
```

Example 29: Specifying a buffer for I/O

An important consideration for C++ I/O in Catamount applications is that the `endl` function causes the data in the buffer to be flushed. In most cases, the `endl` function is used to output a new line, so an `endl` function usually can be replaced in the code by specifying a newline character. In this example, `endl` is redefined to be `'\n'`. If a flush is needed, you can include a call to the `flush()` member function.

Module required:

```
xtpe-target-catamount
```

Source code of `io1.C`

```
#include <iostream>
#include <catamount/dclock.h>

using namespace std;
```

```
#define endl '\n'

int main(int argc, char ** argv) {
  double start, end;
  char *buffer;

  buffer = (char *)malloc(sizeof(char)*12000);
  cout.rdbuf()->pubsetbuf(buffer,12000);
  start = dclock();
  for (int i = 0; i < 1000; i++) {
      cout << "line: " << i << endl;
  }
  end = dclock();
  cout.flush();  // Force a flush of data (not necessary)
  cerr << "Time to write using buffer = " << end - start << endl;

  return 0;
}
```

Compile `io1.C`:

```
% CC -o io1 io1.C
```

Run `io1`, directing output to file `tmp`:

```
% yod io1 > tmp
% cat tmp
Time to write using buffer = 0.000599465
```

Example 30: Changing default buffer size for I/O to file streams

This example uses a default buffer and a modified buffer to write data and prints the time-to-write value for each process.

Module required:

```
xtpe-target-catamount
```

Source code of `io2.C`

```
#include <iostream>
#include <fstream>
#include <catamount/dclock.h>
using namespace std;
#define endl '\n'
```

```
char data[] = " 23456789012345678901234567890 \
01234567890123456789012345678901";

int main(int argc, char ** argv) {
  double start, end;
  char *buffer;

  // Use default buffer
  ofstream data1("output1");
  start = dclock();
  for (int i = 0; i <  10000; i++) {
    data1 <<  "line: " <<  i <<  data <<  endl;
  }
  end = dclock();
  data1.flush(); // Force a flush of data (not necessary)
  cerr <<  "Time to write using default buffer = " \
  << end - start << endl ;

  // Set up a buffer
  ofstream data2("output2");
  buffer = (char *)malloc(sizeof(char)*500000);
  data2.rdbuf()->pubsetbuf(buffer,500000);
  start = dclock();
  for (int i = 0; i < 10000; i++) {
    data2 << "line: " << i << data << endl;
  }
  end = dclock();
  data2.flush(); // Force a flush of data (not necessary)
  cerr << "Time to write with program buffer = " \
  << end - start << endl ;

  return 0;
}
```

Compile `io2.C`:

```
% CC -o io2 io2.C
```

Run `io2`:

```
% yod io2
Time to write using default buffer = 0.0128506
Time to write with program buffer = 0.0237463
```

Example 31: Improving performance of `stdout`

The following program improves the performance of the `printf()` loop by using `setvbuf()` with the mode of `_IOFBF` (fully buffered) and a buffer size of 1024:

Module required:

```
xtpe-target-catamount
```

Source code of C program (`setvbuf1.c`):

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i,bsize,count;
  char *buf;

  i=1;
  bsize = (i<argc) ? atoi(argv[i++]) : 1024;
  count = (i<argc) ? atoi(argv[i++]) : 1024;

  if(bsize > 0) {
    buf = malloc(bsize);
    setvbuf(stdout, buf, _IOFBF, bsize);
  }

  for(i=0;i<count;i++) {
    printf("this is line %5d\n",i);
  }

  exit(0);
}
```

Compile `setvbuf1.c` and create executable `setvbuf1`:

```
% cc -o setvbuf1 setvbuf1.c
```

Run `setvbuf1`:

```
% yod setvbuf1
this is line     0
this is line     1
```

```
this is line     2
this is line     3
...
this is line  1021
this is line  1022
this is line  1023
```

Example 32: Using a PBS Pro job script

This example of a job script, `script1`, requests four processors to run application `program1` (see Example 23, page 136).

Modules required:

```
xtpe-target-catamount
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `script1`.

```
% cat script1
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=4
# Tell PBS Pro to keep both standard output and
# standard error on the execution host:
#PBS -k eo
yod -sz 4 program1
exit 0
```

Set permissions to executable:

```
% chmod +x script1
```

Submit the job:

```
% qsub script1
```

The `qsub` command produces a batch job log file with output from `program1`. The job log file has the form `script1.o`*nnnnnn*.

```
% cat script1.o4595
 hello from pe            0  of           4
```

```
 hello from pe            3  of            4
 hello from pe            2  of            4
 hello from pe            1  of            4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
```

Example 33: Running an MPI program under PBS Pro

This example shows a batch script that runs the program `simple.c` (see Example 21, page 133).

Modules required:

```
xtpe-target-catamount
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create `script2`:

```
% cat script2
#PBS -N s_job
#PBS -l mppwidth=6
#PBS -joe
cd $PBS_O_WORKDIR
yod -sz 6 simple
```

Submit the script to the PBS Pro batch system:

```
% qsub script2
```

Display the job results:

```
% cat s_job.o4596
hello from pe 0 of 6
hello from pe 3 of 6
hello from pe 2 of 6
hello from pe 5 of 6
hello from pe 1 of 6
hello from pe 4 of 6
```

Example 34: Running an MPI_REDUCE program under PBS Pro

This example shows a batch script that runs the program `reduce.f90` (Example 22, page 134).

Modules required:

```
xtpe-target-catamount
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create a batch script, `run_reduce`, verifying that the executable is in a directory in the Lustre file system (see Section 2.4, page 11):

```
% cat run_reduce
#!/bin/sh
#PBS -l mppwidth=2
#PBS -joe
#PBS -l walltime=00:30:00
cd $HOME/pe_user/
echo "Running the Example reduce "
echo ""
date
echo ""
yod -sz 2 reduce
```

set permissions to executable:

```
% chmod +x run_reduce
```

Submit the script to the PBS Pro batch system:

```
% qsub run_reduce
```

Display the job results:

```
% cat run_reduce.o70977
Running the Example reduce

Wed May  9 13:36:52 CDT 2007

 My PE:            1  My part:         2500
 My PE:            0  My part:         2550
    PE:            0 Total is:         5050
```

Example 35: Using a script to create and run a batch job

This example script takes two arguments, the name of a program (`shmem2`, see Example 26, page 141) and the number of processors on which to run the program. The script performs the following actions:

1. Creates a temporary file that contains a PBS Pro batch job script
2. Submits the file to PBS Pro
3. Deletes the temporary file

Modules required:

```
xtpe-target-catamount
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create script `run123`:

```
% cat run123
#!/bin/csh
  if ( "$1" == "" ) then
    echo "Usage: run [executable|script] [ncpus]"
    exit
  endif
  set n=1            # set default number of CPUs
  if ( "$2" != "" ) set n=$2
  cat > job.$$ <<EOT  #creates the batch jobscript
#!/bin/csh
#PBS -N $1
#PBS -l mppwidth=$n
#PBS -joe
cd \$PBS_O_WORKDIR
yod -sz $n -tlimit 30 $1
EOT
qsub job.$$        # submit batch job
rm job.$$
```

Set file permissions to executable:

```
% chmod +x run123
```

Run the job script:

```
% ./run123 shmem2 4
```

List the job output:

```
% cat shmem2.o4611
 PE            1  computedsum=    6.000000
 PE            0  computedsum=    6.000000
 PE            3  computedsum=    6.000000
 PE            2  computedsum=    6.000000
```

Example 36: Running multiple sequential applications

To run multiple sequential applications, the number of processors you specify as an argument to `qsub` must be equal to or greater than the **largest number** of processors required by an invocation of `yod` in your script. For example, in job script `mult_seq_qk`, the `-l mppwidth` is 4 because the largest `yod sz` value is 4.

Modules required:

```
xtpe-target-catamount
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create script `mult_seq_qk`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=4
# Tell PBS Pro to keep both standard output and
# standard error on the execution host:
#PBS -k eo
yod -sz 2 program1
yod -sz 3 program2
yod -sz 4 shmem1
yod -sz 2 shmem2
exit 0
```

The script launches applications `program1` (see Example 23, page 136), `program2` (see Example 24, page 137), `shmem1` (see Example 25, page 139), and `shmem2` (see Example 26, page 141).

Set file permissions to executable:

```
% chmod +x mult_seq_qk
```

Run the script:

```
% qsub mult_seq_qk
```

List the output:

```
% cat mult_seq_qk.o4618
 hello from pe            0  of            2
 hello from pe            1  of            2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
 PE           0 : The answer is:        -1184
 PE 2: Test passed.
 PE 3: Test passed.
 PE 0: Test passed.
 PE 1: Test passed.
 PE           1  computedsum=    1.000000
 PE           0  computedsum=    1.000000
```

Example 37: Running multiple parallel applications

If you are running multiple parallel applications, the number of processors must be equal to or greater than the **total** number of processors specified by calls to `yod`. For example, in job script `mult_par_qk`, the `-l mppwidth` value is `11` because the total of the `yod sz` values is `11`.

Modules required:

```
xtpe-target-catamount
pbs
```

Do not load the `xt-pbs` module. Unload it if it has been loaded.

Create script `mult_par_qk`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=11
# Tell PBS Pro to keep both standard output and
# standard error on the execution host:
#PBS -k eo
yod -sz 2 program1 &
yod -sz 3 program2 &
```

```
yod -sz 4 shmem1 &
yod -sz 2 shmem2 &
exit 0
```

The script launches applications `program1` (see Example 23, page 136), `program2` (see Example 24, page 137), `shmem1` (see Example 25, page 139), and `shmem2` (see Example 26, page 141).

Set file permissions to executable:

```
% chmod +x mult_par_qk
```

Run the script:

```
% qsub mult_par_qk
```

List the output:

```
% cat  mult_par_qk.o13422
 hello from pe 0 of 2
 hello from pe 1 of 2
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
PE 0 : The answer is: -1184
PE 0: Test passed.
PE 3: Test passed.
PE 2: Test passed.
PE 1: Test passed.
PE 0 computedsum= 1.000000
PE 1 computedsum= 1.000000
```

Example 38: Using `xtgdb` to debug a program

This example uses the GNU debugger, `xtgdb`, to debug a program.

Modules required:

```
xtpe-target-catamount
xtgdb
```

Compile program `hi.c`:

```
% cc -g hi.c
```

Initiate a PBS Pro interactive session:

```
% qsub -I
```

Run xtgdb:

```
% xtgdb yod a.out
  Debugging a.out
  Target port is 33381

  Please wait while connecting to catamount...

  target remote :33381
  Remote debugging using :33381
  0x0000000000200001 in _start ()
```

Set breakpoints, resume execution, and quit the gdb session:

```
(gdb) b main

Breakpoint 3 at 0x205674: file hi.c, line 3.

(gdb) c

Continuing.

  Breakpoint 3, main () at hi.c:3
  3       printf("hello.c\n");

  (gdb) c

Continuing.
  hello.c

  Program exited with code 0377.

(gdb) quit

Done
```

Example 39: Using the high-level PAPI interface

PAPI provides simple high-level interfaces for instrumenting applications written in C or Fortran. This example shows the use of the PAPI_start_counters() and PAPI_stop_counters() functions.

Modules required:

```
xtpe-target-catamount
papi
```

Source code of `papi_hl.c`:

```
#include <papi.h>
void main()
{
  int retval, Events[2]= {PAPI_TOT_CYC, PAPI_TOT_INS};
  long_long values[2];

  if (PAPI_start_counters (Events, 2) != PAPI_OK) {
    printf("Error starting counters\n");
    exit(1);
  }

  /* Do some computation here... */

  if (PAPI_stop_counters (values, 2) !=  PAPI_OK) {
    printf("Error stopping counters\n");
    exit(1);
  }

  printf("PAPI_TOT_CYC = %lld\n", values[0]);
  printf("PAPI_TOT_INS = %lld\n", values[1]);
}
```

Compile `papi_hl.c`:

```
% cc -o papi_hl papi_hl.c
```

Run `papi_hl`:

```
% yod papi_hl
PAPI_TOT_CYC = 3287
PAPI_TOT_INS = 287
```

Example 40: Using the low-level PAPI interface

PAPI provides an advanced low-level interface for instrumenting applications. The PAPI library must be initialized before calling any of these functions; initialization can be done by issuing either a high-level function call or a call to `PAPI_library_init()`. This example shows the use of the

PAPI_create_eventset(), PAPI_add_event()), PAPI_start(), and PAPI_read() functions.

Modules required:

```
xtpe-target-catamount
papi
```

Source code of papi_ll.c:

```
#include <papi.h>
void main()
{
  int EventSet = PAPI_NULL;
  long_long values[1];

  /* Initialize PAPI library */
  if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
    printf("Error initializing PAPI library\n");
    exit(1);
  }

  /* Create Event Set */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK) {
    printf("Error creating eventset\n");
    exit(1);
  }

  /* Add Total Instructions Executed to eventset */
  if (PAPI_add_event (EventSet, PAPI_TOT_INS) != PAPI_OK) {
    printf("Error adding event\n");
    exit(1);
  }

  /* Start counting ... */
  if (PAPI_start (EventSet) != PAPI_OK) {
    printf("Error starting counts\n");
    exit(1);
  }

  /* Do some computation here...*/

  if (PAPI_read (EventSet, values) != PAPI_OK) {
    printf("Error stopping counts\n");
```

```
    exit(1);
  }
  printf("PAPI_TOT_INS = %lld\n", values[0]);
}
```

Compile `papi_ll.c`:

```
% cc -o papi_ll papi_ll.c
```

Run `papi_ll`:

```
% yod papi_ll
PAPI_TOT_INS = 153
```

Example 41: Using basic CrayPat functions

This example shows how to instrument a program, run the instrumented program, and generate CrayPat reports.

Modules required:

```
xtpe-target-catamount
craypat
```

Compile the sample program `prog.f90` and the routine it calls, `work.c`.

Source code of `prog.f90`:

```
program main
include 'mpif.h'

  call MPI_Init(ierr)     ! Required
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  print *,'hello from pe',mype,' of',npes

  do i=1+mype,1000,npes   ! Distribute the work
    call work(i,mype)
  enddo

  call MPI_Finalize(ierr) ! Required
end
```

Source code of `work.c`:

```
void work_(int *N, int *MYPE)
```

```
{
  int n=*N, mype=*MYPE;

  if (n == 42) {
    printf("PE %d: sizeof(long) = %d\n",mype,sizeof(long));
    printf("PE %d: The answer is: %d\n",mype,n);
  }
}
```

Compile `prog.f90` and `work.c` and create executable `program1`:

```
% cc -c work.c
% ftn -o program1 prog.f90 work.o
```

Run `pat_build` to generate instrumented program `program1+pat`:

```
% pat_build -u -g mpi program1 program1+pat
INFO: A trace intercept routine was created for the function 'work_'.
INFO: a total of 39 function entry points were traced
```

The tracegroup (`-g` option) is `mpi`.

Set environment variable `PAT_RT_EXPFILE_PER_PROCESS`:

```
% setenv PAT_RT_EXPFILE_PER_PROCESS 1
```

Run `program1+pat`:

```
% yod -sz 4 program1+pat
CrayPat/X:  Version 3.2 Revision 799  04/23/07 08:02:31
 hello from pe            3  of            4
 hello from pe            1  of            4
 hello from pe            2  of            4
 hello from pe            0  of            4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Experiment data file written:
/lus/nid00007/user1/catamount/program1+pat+87td.xf
```

Note: When executed, the instrumented executable creates directory *progname*`+pat+`*PIDkeyletters* that contains one or more data files with a `.xf` suffix. *PID* is the process ID that was assigned to the instrumented program at run time.

Run `pat_report` to generate reports `program1.rpt1` (using default `pat_report` options) and `program1.rpt2` (using the `-O calltree` option).

```
% pat_report program1+pat+87td.xf > program1.rpt1
Data file 4/4: [....................]
% pat_report -O calltree program1+pat+87td.xf > program1.rpt2
Data file 4/4: [....................]
```

List `program1.rpt1`:

```
CrayPat/X:  Version 3.2 Revision 799 (xf 784)  04/23/07 08:02:31

Experiment:  trace

Experiment data file:
  /lus/nid00007/user1/catamount/program1+pat+87td.xf  (RTS)

Original program:  /lus/nid00007/user1/catamount/program1

Instrumented with:  pat_build -u -g mpi program1 program1+pat

Instrumented program:  /lus/nid00007/user1/catamount/program1+pat

Program invocation:  program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/2.0.06/mpich2-64
  MPICH_DIR=/opt/xt-mpt/2.0.06/mpich2-64/P2
  MPICH_DIR_FTN_DEFAULT64=/opt/xt-mpt/2.0.06/mpich2-64/P2W

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx

Report command line options:  <none>

System name, type, and speed:  xt1 x86_64  2400 MHz

Operating system:  catamount 1.0 2.0

Estimated minimum overhead per call of a traced function,
```

```
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    Time    0.617  microseconds

Number of traced functions: 52

Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
    -d ti%@0.05,ti,imb_ti,imb_ti%,tr -b gr,fu,pe=HIDE

  Options for related tables not shown by default:
    -O load_balance
    -O callers
    -O callers+src
    -O calltree
    -O calltree+src

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).

Table 1:  Profile by Function Group and Function

 Time % |     Time |Imb. Time |   Imb. | Calls |Group
        |          |          | Time % |       | Function
        |          |          |        |       |  PE='HIDE'

 100.0% | 0.003184 |       -- |     -- |  2628 |Total
|-----------------------------------------------------------
|  98.1% | 0.003124 |       -- |     -- |  1012 |USER
||----------------------------------------------------------
||  97.0% | 0.003031 | 0.000113 |   4.8% |     4 |MAIN_
||   2.3% | 0.000070 | 0.000193 |  97.7% |  1000 |work_
||   0.7% | 0.000021 | 0.000000 |   0.9% |     4 |exit
||   0.1% | 0.000002 | 0.000000 |   4.0% |     4 |main
||==========================================================
|   0.1% | 0.000002 |       -- |     -- |    16 |MPI
||----------------------------------------------------------
||  31.5% | 0.000001 | 0.000000 |   7.3% |     4 |mpi_init_
```

```
||  24.1% | 0.000000 | 0.000000 |   8.4% |     4 |mpi_comm_rank_
||  23.6% | 0.000000 | 0.000000 |   5.7% |     4 |mpi_comm_size_
||  20.8% | 0.000000 | 0.000000 |  22.3% |     4 |mpi_finalize_
|=============================================================

<snip>

Table 3:  Program Wall Clock Time

  Process |PE
     Time |

 0.256492 |Total
|------------
| 0.280461 |pe.1
| 0.264507 |pe.0
| 0.248539 |pe.2
| 0.232462 |pe.3
|===========
```

List `program1.rpt2`:

```
CrayPat/X:  Version 3.2 Revision 799 (xf 784)  04/23/07 08:02:31

Experiment:  trace

Experiment data file:
  /lus/nid00007/user1/catamount/program1+pat+87td.xf  (RTS)

Original program:  /lus/nid00007/user1/catamount/program1

Instrumented with:  pat_build -u -g mpi program1 program1+pat

Instrumented program:  /lus/nid00007/user1/catamount/program1+pat

Program invocation:  program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/2.0.06/mpich2-64
  MPICH_DIR=/opt/xt-mpt/2.0.06/mpich2-64/P2
```

```
  MPICH_DIR_FTN_DEFAULT64=/opt/xt-mpt/2.0.06/mpich2-64/P2W

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.2.1/cpatx

Report command line options:  -O calltree

System name, type, and speed:  xt1 x86_64  2400 MHz

Operating system:  catamount 1.0 2.0

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    Time    0.617  microseconds

Number of traced functions: 52

Notes for table 1:

  Table option:
    -O calltree
  Options implied by table option:
    -d ti%@0.05,cum_ti%,ti,tr -b ct,pe=HIDE

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).

Table 1:  Function Calltree View

 Time % |   Cum. |     Time | Calls |Calltree
        | Time % |          |       | PE='HIDE'

 100.0% | 100.0% | 0.003184 |  2628 |Total
|-----------------------------------------------
|  98.2% |  98.2% | 0.003126 |  1028 |main
||----------------------------------------------
||  99.3% |  99.3% | 0.003104 |  1020 |MAIN_
|||---------------------------------------------
3||  97.7% |  97.7% | 0.003031 |     4 |MAIN_(exclusive)
3||   2.3% |  99.9% | 0.000070 |  1000 |work_
```

```
|||==========================================
||   0.7% |  99.9% | 0.000021 |     4 |exit
||   0.1% | 100.0% | 0.000002 |     4 |main(exclusive)
||==========================================
|   1.3% |  99.5% | 0.000042 |   800 |__do_global_ctors
|   0.5% | 100.0% | 0.000016 |   800 |exit
|==========================================
```

Example 42: Using hardware performance counters

This example uses the same instrumented program as Example 41, page 158 and generates reports showing hardware performance counter (HWPC) information.

Modules required:

```
xtpe-target-catamount
craypat
```

Collect HWPC event set 1 information and generate report `program1.rpt3` (for a list of predefined event sets, see the `hwpc`(3) man page):

```
% setenv PAT_RT_HWPC 1
% yod -sz 4 program1+pat
CrayPat/X:  Version 3.1 Revision 363  08/28/06 16:25:58
 hello from pe            3  of            4
 hello from pe            1  of            4
 hello from pe            2  of            4
 hello from pe            0  of            4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Experiment data directory written:
/ufs/home/users/user1/pat/program1+pat+2518td
% pat_report program1+pat+2518td > program1.rpt3
Data file 4/4:
[...................]
```

List `program1.rpt3`:

```
CrayPat/X:  Version 3.1 Revision 609 (xf 556)  01/23/07 11:48:46

Experiment:  trace

Experiment data file:
  /ufs/home/users/user1/guide_test/program1+pat+142td/*.xf  (RTS)

Original program:  /ufs/home/users/user1/guide_test/program1
```

```
Instrumented program:  /ufs/home/users/user1/guide_test/program1+pat

Program invocation:  program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/1.4.48/mpich2-64
  MPICH_DIR=/opt/xt-mpt/1.4.48/mpich2-64/P2
  PAT_BUILD_ASYNC=0
  PAT_ROOT=/opt/xt-tools/craypat/3.1.2/cpatx
  PAT_RT_EXPFILE_PER_PROCESS=1
  PAT_RT_HWPC=1

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1.2/cpatx

Report command line options:  <none>

Host name and type:  sys1 x86_64  2400 MHz

Operating system:  catamount 1.0 2.0

Hardware performance counter events:
  PAPI_TLB_DM  Data translation lookaside buffer misses
  PAPI_L1_DCA  Level 1 data cache accesses
  PAPI_FP_OPS  Floating point operations
  DC_MISS      Data Cache Miss
  User_Cycles  Virtual Cycles

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    PAPI_TLB_DM      5.000  misses
    PAPI_L1_DCA   1318.298  ops
    PAPI_FP_OPS      0.000  ops
    DC_MISS          4.509  ops
    User_Cycles   2105.166  cycles
    Time             0.877  microseconds
```

```
Traced functions:
  MAIN_                     .../users/user1/guide_test/prog.f90
  MPI_Abort                 ==NA==
<snip>
  work_                     .../users/user1/guide_test/work.c



Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
    -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P -b ex,gr,fu,pe=HIDE

  Options for related tables not shown by default:
    -O load_balance
    -O callers
    -O callers+src
    -O calltree
    -O calltree+src

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Profile by Function Group and Function

Experiment=1 / Group / Function / PE='HIDE'

======================================================================
Totals for program
----------------------------------------------------------------------
  Time%                                   100.0%
  Time                                 0.002658
  Imb.Time                                   --
  Imb.Time%                                  --
  Calls                                 17028
  PAPI_TLB_DM              24.674M/sec     66159 misses
  PAPI_L1_DCA            5042.230M/sec  13519803 ops
  PAPI_FP_OPS               0.183M/sec       490 ops
```

```
  DC_MISS                  22.031M/sec     59073 ops
  User time                 0.003 secs   6435154 cycles
  Utilization rate                        100.0%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time     0.183M/sec       490 ops        0.0%peak
  HW FP Ops / WCT           0.183M/sec
  Computation intensity                     0.00 ops/ref
  LD & ST per TLB miss                    204.35 ops/miss
  LD & ST per D1 miss                     228.87 ops/miss
  D1 cache hit ratio                       99.6%
  % TLB misses / cycle                      0.3%
=============================================================================
USER
-----------------------------------------------------------------------------
  Time%                                    62.7%
  Time                                 0.001665
  Imb.Time                                   --
  Imb.Time%                                  --
  Calls                                  1012
  PAPI_TLB_DM              15.488M/sec     25796 misses
  PAPI_L1_DCA            4702.512M/sec   7832225 ops
  PAPI_FP_OPS               0.294M/sec       490 ops
  DC_MISS                   6.107M/sec     10172 ops
  User time                 0.002 secs   3997298 cycles
  Utilization rate                        100.0%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time     0.294M/sec       490 ops        0.0%peak
  HW FP Ops / WCT           0.294M/sec
  Computation intensity                     0.00 ops/ref
  LD & ST per TLB miss                    303.62 ops/miss
  LD & ST per D1 miss                     769.98 ops/miss
  D1 cache hit ratio                       99.9%
  % TLB misses / cycle                      0.2%
=============================================================================
USER / work_
-----------------------------------------------------------------------------
  Time%                                    43.4%
  Time                                 0.000723
  Imb.Time                             0.002141
  Imb.Time%                                99.7%
  Calls                                  1000
  PAPI_TLB_DM               0.291M/sec       211 misses
  PAPI_L1_DCA            4228.537M/sec   3061262 ops
```

```
  PAPI_FP_OPS                                0 ops
  DC_MISS                  0.724M/sec      524 ops
  User time                0.001 secs  1737487 cycles
  Utilization rate                      100.0%
  HW FP Ops / Cycles                      0.00 ops/cycle
  HW FP Ops / User time                      0 ops        0.0%peak
  HW FP Ops / WCT
  Computation intensity                   0.00 ops/ref
  LD & ST per TLB miss                14508.35 ops/miss
  LD & ST per D1 miss                  5842.10 ops/miss
  D1 cache hit ratio                    100.0%
  % TLB misses / cycle                    0.0%
=========================================================================
USER / MAIN_
-------------------------------------------------------------------------
  Time%                                  31.4%
  Time                                0.000523
  Imb.Time                            0.000098
  Imb.Time%                              21.0%
  Calls                                     4
  PAPI_TLB_DM             10.621M/sec     5527 misses
  PAPI_L1_DCA           4481.995M/sec  2332287 ops
  PAPI_FP_OPS              0.411M/sec      214 ops
  DC_MISS                  6.378M/sec     3319 ops
  User time                0.001 secs  1248883 cycles
  Utilization rate                       99.5%
  HW FP Ops / Cycles                      0.00 ops/cycle
  HW FP Ops / User time    0.411M/sec      214 ops        0.0%peak
  HW FP Ops / WCT          0.409M/sec
  Computation intensity                   0.00 ops/ref
  LD & ST per TLB miss                  421.98 ops/miss
  LD & ST per D1 miss                   702.71 ops/miss
  D1 cache hit ratio                     99.9%
  % TLB misses / cycle                    0.1%
=========================================================================
USER / exit
-------------------------------------------------------------------------
  Time%                                  25.1%
  Time                                0.000417
  Imb.Time                            0.000015
  Imb.Time%                               4.5%
  Calls                                     4
  PAPI_TLB_DM             47.731M/sec    20026 misses
```

```
  PAPI_L1_DCA            5805.125M/sec   2435599 ops
  PAPI_FP_OPS               0.648M/sec       272 ops
  DC_MISS                  14.913M/sec      6257 ops
  User time                 0.000 secs   1006944 cycles
  Utilization rate                        100.0%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time     0.648M/sec       272 ops        0.0%peak
  HW FP Ops / WCT           0.648M/sec
  Computation intensity                     0.00 ops/ref
  LD & ST per TLB miss                    121.62 ops/miss
  LD & ST per D1 miss                     389.26 ops/miss
  D1 cache hit ratio                       99.7%
  % TLB misses / cycle                      0.5%
=========================================================================
USER / main
-------------------------------------------------------------------------
  Time%                                     0.1%
  Time                                  0.000002
  Imb.Time                              0.000000
  Imb.Time%                                 2.3%
  Calls                                        4
  PAPI_TLB_DM              19.281M/sec        32 misses
  PAPI_L1_DCA            1853.963M/sec      3077 ops
  PAPI_FP_OPS               2.410M/sec         4 ops
  DC_MISS                  43.382M/sec        72 ops
  User time                 0.000 secs      3983 cycles
  Utilization rate                         95.3%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time     2.410M/sec         4 ops        0.0%peak
  HW FP Ops / WCT           2.298M/sec
  Computation intensity                     0.00 ops/ref
  LD & ST per TLB miss                     96.16 ops/miss
  LD & ST per D1 miss                      42.74 ops/miss
  D1 cache hit ratio                       97.7%
  % TLB misses / cycle                      0.2%
=========================================================================
MPI
-------------------------------------------------------------------------
  Time%                                     0.1%
  Time                                  0.000003
  Imb.Time                                    --
  Imb.Time%                                   --
  Calls                                       16
```

```
  PAPI_TLB_DM              18.966M/sec        51 misses
  PAPI_L1_DCA            3298.175M/sec      8869 ops
  PAPI_FP_OPS                                  0 ops
  DC_MISS                  68.053M/sec       183 ops
  User time                 0.000 secs      6454 cycles
  Utilization rate                         97.3%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time                        0 ops        0.0%peak
  HW FP Ops / WCT
  Computation intensity                     0.00 ops/ref
  LD & ST per TLB miss                    173.90 ops/miss
  LD & ST per D1 miss                      48.46 ops/miss
  D1 cache hit ratio                       97.9%
  % TLB misses / cycle                      0.2%
=============================================================================
MPI / mpi_comm_size_
-----------------------------------------------------------------------------
  Time%                                    28.8%
  Time                                 0.000001
  Imb.Time                             0.000000
  Imb.Time%                                8.9%
  Calls                                       4
  PAPI_TLB_DM              13.741M/sec        11 misses
  PAPI_L1_DCA            2503.370M/sec      2004 ops
  PAPI_FP_OPS                                  0 ops
  DC_MISS                  58.712M/sec        47 ops
  User time                 0.000 secs      1921 cycles
  Utilization rate                        100.0%
  HW FP Ops / Cycles                        0.00 ops/cycle
  HW FP Ops / User time                        0 ops        0.0%peak
  HW FP Ops / WCT
  Computation intensity                     0.00 ops/ref
  LD & ST per TLB miss                    182.18 ops/miss
  LD & ST per D1 miss                      42.64 ops/miss
  D1 cache hit ratio                       97.7%
  % TLB misses / cycle                      0.1%
=============================================================================
MPI / mpi_init_
-----------------------------------------------------------------------------
  Time%                                    24.1%
  Time                                 0.000001
  Imb.Time                             0.000000
  Imb.Time%                                10.7%
```

```
  Calls                                      4
  PAPI_TLB_DM             13.413M/sec        8 misses
  PAPI_L1_DCA           4590.430M/sec     2738 ops
  PAPI_FP_OPS                              0 ops
  DC_MISS                 80.475M/sec       48 ops
  User time                0.000 secs     1432 cycles
  Utilization rate                       89.4%
  HW FP Ops / Cycles                      0.00 ops/cycle
  HW FP Ops / User time                      0 ops        0.0%peak
  HW FP Ops / WCT
  Computation intensity                   0.00 ops/ref
  LD & ST per TLB miss                  342.25 ops/miss
  LD & ST per D1 miss                    57.04 ops/miss
  D1 cache hit ratio                     98.2%
  % TLB misses / cycle                    0.1%
=========================================================================
MPI / mpi_finalize_
-------------------------------------------------------------------------
  Time%                                  24.1%
  Time                                0.000001
  Imb.Time                            0.000000
  Imb.Time%                              13.2%
  Calls                                      4
  PAPI_TLB_DM             21.737M/sec       14 misses
  PAPI_L1_DCA           3372.344M/sec     2172 ops
  PAPI_FP_OPS                              0 ops
  DC_MISS                 74.527M/sec       48 ops
  User time                0.000 secs     1546 cycles
  Utilization rate                       96.5%
  HW FP Ops / Cycles                      0.00 ops/cycle
  HW FP Ops / User time                      0 ops        0.0%peak
  HW FP Ops / WCT
  Computation intensity                   0.00 ops/ref
  LD & ST per TLB miss                  155.14 ops/miss
  LD & ST per D1 miss                    45.25 ops/miss
  D1 cache hit ratio                     97.8%
  % TLB misses / cycle                    0.2%
=========================================================================
MPI / mpi_comm_rank_
-------------------------------------------------------------------------
  Time%                                  22.9%
  Time                                0.000001
  Imb.Time                            0.000000
```

```
  Imb.Time%                                 11.6%
  Calls                                        4
  PAPI_TLB_DM                 27.777M/sec     18 misses
  PAPI_L1_DCA               3016.878M/sec   1955 ops
  PAPI_FP_OPS                                  0 ops
  DC_MISS                     61.726M/sec     40 ops
  User time                    0.000 secs    1555 cycles
  Utilization rate                         100.0%
  HW FP Ops / Cycles                         0.00 ops/cycle
  HW FP Ops / User time                         0 ops        0.0%peak
  HW FP Ops / WCT
  Computation intensity                      0.00 ops/ref
  LD & ST per TLB miss                     108.61 ops/miss
  LD & ST per D1 miss                       48.88 ops/miss
  D1 cache hit ratio                        98.0%
  % TLB misses / cycle                       0.3%
=========================================================================


Notes for table 2:

  Table option:
    -O heap_program
  Options implied by table option:
    -d IU,IF,NF,FM -b ex,pe


Table 2:  Heap Usage at Start and End of Main Program

MB Heap |  MB Heap |  Heap | Max Free |Experiment=1
Used at |  Free at |   Not |Object at |PE
  Start |    Start | Freed |      End |
        |          |    MB |          |

 94.656 | 3875.344 | 0.023 | 3875.321 |Total
|-------------------------------------------------
| 94.660 | 3875.340 | 0.023 | 3875.316 |pe.0
| 94.654 | 3875.346 | 0.023 | 3875.322 |pe.1
| 94.654 | 3875.346 | 0.023 | 3875.322 |pe.3
| 94.654 | 3875.346 | 0.023 | 3875.322 |pe.2
|================================================
```

```
Notes for table 3:

  Table option:
    -O program_time
  Options implied by table option:
    -d pt -b ex,pe


Table 3:  Program Wall Clock Time

  Process |Experiment=1
     Time |PE

 0.014952 |Total
|----------------------
| 0.016712 |pe.1
| 0.016441 |pe.2
| 0.013384 |pe.0
| 0.013271 |pe.3
|======================
```

Collect information about translation lookaside buffer (TLB) misses (PAPI_TLB_DM) and generate report program1.rpt4:

```
% setenv PAT_RT_HWPC PAPI_TLB_DM
% yod -sz 4 program1+pat
 hello from pe            1  of           4
 hello from pe            2  of           4
 hello from pe            3  of           4
 hello from pe            0  of           4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
Experiment data directory written:
/ufs/home/users/user1/pat/program1+pat+2520td
% pat_report program1+pat+2520td > program1.rpt4
Data file 4/4: [.....................]
```

List program1.rpt4:

```
CrayPat/X:  Version 3.1 Revision 609 (xf 556)  01/23/07 11:48:46

Experiment:  trace

Experiment data file:
```

```
  /ufs/home/users/user1/guide_test/program1+pat+143td/*.xf  (RTS)

Original program:  /ufs/home/users/user1/guide_test/program1

Instrumented program:  /ufs/home/users/user1/guide_test/program1+pat

Program invocation:  program1+pat

Number of PEs:  4

Exit Status:  0  PEs:  0-3

Runtime environment variables:
  MPICHBASEDIR=/opt/xt-mpt/1.4.48/mpich2-64
  MPICH_DIR=/opt/xt-mpt/1.4.48/mpich2-64/P2
  PAT_BUILD_ASYNC=0
  PAT_ROOT=/opt/xt-tools/craypat/3.1.2/cpatx
  PAT_RT_EXPFILE_PER_PROCESS=1
  PAT_RT_HWPC=PAPI_TLB_DM

Report time environment variables:
  PAT_ROOT=/opt/xt-tools/craypat/3.1.2/cpatx

Report command line options:  <none>

Host name and type:  sys1 x86_64  2400 MHz

Operating system:  catamount 1.0 2.0

Hardware performance counter events:
  PAPI_TLB_DM  Data translation lookaside buffer misses
  User_Cycles  Virtual Cycles

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    PAPI_TLB_DM      5.000  misses
    User_Cycles   1977.854  cycles
    Time             0.827  microseconds

Traced functions:
  MAIN_                      .../users/user1/guide_test/prog.f90
  MPI_Abort                  ==NA==
```

```
<snip>:
  work_                     .../users/user1/guide_test/work.c



Notes for table 1:

  Table option:
    -O profile
  Options implied by table option:
    -d ti%@0.05,ti,imb_ti,imb_ti%,tr,P -b ex,gr,fu,pe=HIDE

  Options for related tables not shown by default:
    -O load_balance
    -O callers
    -O callers+src
    -O calltree
    -O calltree+src

  This table shows only lines with Time% > 0.05.

  Percentages at each level are relative
    (for absolute percentages, specify:  -s percent=a).


Table 1:  Profile by Function Group and Function

Experiment=1 / Group / Function / PE='HIDE'

=============================================================================
Totals for program
-----------------------------------------------------------------------------
  Time%                                  100.0%
  Time                                0.002753
  Imb.Time                                  --
  Imb.Time%                                 --
  Calls                                  17028
  PAPI_TLB_DM           24.252M/sec      67725 misses
  User time              0.003 secs    6702061 cycles
  Utilization rate                       100.0%
  % TLB misses / cycle                     0.3%
=============================================================================
USER
```

```
------------------------------------------------------------------------
  Time%                                  68.5%
  Time                                0.001885
  Imb.Time                                --
  Imb.Time%                               --
  Calls                                 1012
  PAPI_TLB_DM           13.745M/sec     25902 misses
  User time              0.002 secs   4522640 cycles
  Utilization rate                     100.0%
  % TLB misses / cycle                   0.1%
========================================================================
USER / MAIN_
------------------------------------------------------------------------
  Time%                                  41.7%
  Time                                0.000786
  Imb.Time                            0.000098
  Imb.Time%                              14.7%
  Calls                                     4
  PAPI_TLB_DM           7.102M/sec       5570 misses
  User time             0.001 secs   1882248 cycles
  Utilization rate                       99.8%
  % TLB misses / cycle                   0.1%
========================================================================
USER / work_
------------------------------------------------------------------------
  Time%                                  38.7%
  Time                                0.000730
  Imb.Time                            0.002164
  Imb.Time%                              99.7%
  Calls                                  1000
  PAPI_TLB_DM           0.383M/sec        280 misses
  User time             0.001 secs   1753760 cycles
  Utilization rate                     100.0%
  % TLB misses / cycle                   0.0%
========================================================================
USER / exit
------------------------------------------------------------------------
  Time%                                  19.5%
  Time                                0.000367
  Imb.Time                            0.000011
  Imb.Time%                              3.8%
  Calls                                     4
  PAPI_TLB_DM           54.438M/sec     20023 misses
```

```
  User time              0.000 secs    882755 cycles
  Utilization rate                     100.0%
  % TLB misses / cycle                   0.6%
==========================================================================
USER / main
--------------------------------------------------------------------------
  Time%                                  0.1%
  Time                                0.000002
  Imb.Time                            0.000000
  Imb.Time%                              2.9%
  Calls                                     4
  PAPI_TLB_DM           17.953M/sec        29 misses
  User time              0.000 secs      3877 cycles
  Utilization rate                      97.4%
  % TLB misses / cycle                   0.2%
==========================================================================
MPI
--------------------------------------------------------------------------
  Time%                                  0.1%
  Time                                0.000003
  Imb.Time                                  --
  Imb.Time%                                 --
  Calls                                    16
  PAPI_TLB_DM           14.478M/sec        38 misses
  User time              0.000 secs      6299 cycles
  Utilization rate                      95.2%
  % TLB misses / cycle                   0.2%
==========================================================================
MPI / mpi_comm_size_
--------------------------------------------------------------------------
  Time%                                 34.7%
  Time                                0.000001
  Imb.Time                            0.000000
  Imb.Time%                              8.7%
  Calls                                     4
  PAPI_TLB_DM           12.902M/sec        12 misses
  User time              0.000 secs      2232 cycles
  Utilization rate                      97.1%
  % TLB misses / cycle                   0.1%
==========================================================================
MPI / mpi_init_
--------------------------------------------------------------------------
  Time%                                 24.0%
```

```
  Time                               0.000001
  Imb.Time                           0.000000
  Imb.Time%                             11.8%
  Calls                                    4
  PAPI_TLB_DM           7.078M/sec         4 misses
  User time             0.000 secs      1356 cycles
  Utilization rate                      85.5%
  % TLB misses / cycle                   0.1%
=============================================================================
MPI / mpi_finalize_
-----------------------------------------------------------------------------
  Time%                                 22.9%
  Time                               0.000001
  Imb.Time                           0.000000
  Imb.Time%                             11.8%
  Calls                                    4
  PAPI_TLB_DM          14.037M/sec         9 misses
  User time             0.000 secs      1539 cycles
  Utilization rate                     100.0%
  % TLB misses / cycle                   0.1%
=============================================================================
MPI / mpi_comm_rank_
-----------------------------------------------------------------------------
  Time%                                 18.3%
  Time                               0.000001
  Imb.Time                           0.000000
  Imb.Time%                              9.4%
  Calls                                    4
  PAPI_TLB_DM          26.627M/sec        13 misses
  User time             0.000 secs      1172 cycles
  Utilization rate                      96.5%
  % TLB misses / cycle                   0.3%
=============================================================================


Notes for table 2:

  Table option:
    -O heap_program
  Options implied by table option:
    -d IU,IF,NF,FM -b ex,pe
```

```
Table 2:  Heap Usage at Start and End of Main Program

MB Heap |  MB Heap |  Heap | Max Free |Experiment=1
Used at |  Free at |   Not |Object at |PE
  Start |    Start | Freed |      End |
        |          |    MB |          |

 94.656 | 3875.344 | 0.023 | 3875.321 |Total
|-------------------------------------------------
| 94.660 | 3875.340 | 0.023 | 3875.316 |pe.0
| 94.654 | 3875.346 | 0.023 | 3875.322 |pe.1
| 94.654 | 3875.346 | 0.023 | 3875.322 |pe.3
| 94.654 | 3875.346 | 0.023 | 3875.322 |pe.2
|================================================


Notes for table 3:

  Table option:
    -O program_time
  Options implied by table option:
    -d pt -b ex,pe


Table 3:  Program Wall Clock Time

  Process |Experiment=1
     Time |PE

 0.014993 |Total
|----------------------
| 0.018695 |pe.1
| 0.013868 |pe.2
| 0.013706 |pe.0
| 0.013704 |pe.3
|======================
```

glibc Functions Supported in CNL [A]

The glibc functions and system calls supported in CNL are listed in Table 9. For further information, see the man pages.

Note: Some `fcntl()` commands are not supported for applications that use Lustre. The supported commands are:

- `F_GETFL`
- `F_SETFL`
- `F_GETLK`
- `F_SETLK`
- `F_SETLKW64`
- `F_SETLKW`
- `F_SETLK64`

Table 9. Supported glibc Functions for CNL

a64l	abort	abs	access
addmntent	alarm	alphasort	argz_add
argz_add_sep	argz_append	argz_count	argz_create
argz_create_sep	argz_delete	argz_extract	argz_insert
argz_next	argz_replace	argz_stringify	asctime
asctime_r	asprintf	atexit	atof
atoi	atol	atoll	basename
bcmp	bcopy	bind_textdomain_codeset	bindtextdomain
bsearch	btowc	bzero	calloc
catclose	catgets	catopen	cbc_crypt
chdir	chmod	chown	clearenv
clearerr	clearerr_unlocked	close	closedir
confstr	copysign	copysignf	copysignl
creat	ctime	ctime_r	daemon

daylight
dcgettext
dcngettext
des_setparity
dgettext
difftime
dirfd
dirname
div
dngettext
dprintf
drand48
dup
dup2
dysize
ecb_crypt
ecvt
ecvt_r
endfsent
endmntent
endttyent
endusershell
envz_add
envz_entry
envz_get
envz_merge
envz_remove
envz_strip
erand48
err
errx
exit
fchmod
fchown
fclose
fcloseall
fcntl
fcvt
fcvt_r
fdatasync
fdopen
feof
feof_unlocked
ferror
ferror_unlocked
fflush
fflush_unlocked
ffs
ffsl
ffsll
fgetc
fgetc_unlocked
fgetgrent
fgetpos
fgetpwent
fgets
fgets_unlocked
fgetwc
fgetwc_unlocked
fgetws
fgetws_unlocked
fileno
fileno_unlocked
finite
flockfile
fnmatch
fopen
fprintf
fputc
fputc_unlocked
fputs
fputs_unlocked
fputwc
fputwc_unlocked
fputws
fputws_unlocked
fread
fread_unlocked
free
freopen
frexp
fscanf
fseek
fseeko
fsetpos
fstat
fsync
ftell
ftello
ftime
ftok
ftruncate
ftrylockfile
funlockfile
fwide
fwprintf
fwrite
fwrite_unlocked
gcvt
get_current_dir_name
getc
getc_unlocked
getchar
getchar_unlocked
getcwd
getdate
getdate_r
getdelim
getdirentries
getdomainname
getegid
getenv
geteuid
getfsent
getfsfile
getfsspec
getgid
gethostname
getline
getlogin

getlogin_r
getmntent
getopt
getopt_long
getopt_long_only
getpagesize
getpass
getpid
getrlimit
getrusage
gettext
gettimeofday
getttyent
getttynam
getuid
getusershell
getw
getwc
getwc_unlocked
getwchar
getwchar_unlocked
gmtime
gmtime_r
gsignal
hasmntopt
hcreate
hcreate_r
hdestroy
hsearch
iconv
iconv_close
iconv_open
imaxabs
index
initstate
insque
ioctl
isalnum
isalpha
isascii
isblank
iscntrl
isdigit
isgraph
isinf
islower
isnan
isprint
ispunct
isspace
isupper
iswalnum
iswalpha
iswblank
iswcntrl
iswctype
iswdigit
iswgraph
iswlower
iswprint
iswpunct
iswspace
iswupper
iswxdigit
isxdigit
jrand48
kill
l64a
labs
lcong48
ldexp
lfind
link
llabs
localeconv
localtime
localtime_r
lockf
longjmp
lrand48
lsearch
lseek
lstat
malloc
mblen
mbrlen
mbrtowc
mbsinit
mbsnrtowcs
mbsrtowcs
mbstowcs
mbtowc
memccpy
memchr
memcmp
memcpy
memfrob
memmem
memmove
memrchr
memset
mkdir
mkdtemp
mknod
mkstemp
mktime
modf
modff
modfl
mrand48
nanosleep
ngettext
nl_langinfo
nrand48
on_exit
open
opendir
passwd2des
pclose
perror

pread	printf	psignal	putc
putc_unlocked	putchar	putchar_unlocked	putenv
putpwent	puts	putw	putwc
putwc_unlocked	putwchar	putwchar_unlocked	pwrite
qecvt	qecvt_r	qfcvt	qfcvt_r
qgcvt	qsort	raise	rand
random	re_comp	re_exec	read
readdir	readlink	readv	realloc
realpath	regcomp	regerror	regexec
regfree	registerrpc	remove	remque
rename	rewind	rewinddir	rindex
rmdir	scandir	scanf	seed48
seekdir	setbuf	setbuffer	setegid
setenv	seteuid	setfsent	setgid
setitimer	setjmp	setlinebuf	setlocale
setlogmask	setmntent	setrlimit	setstate
setttyent	setuid	setusershell	setvbuf
sigaction	sigaction[1]	sigaddset	sigdelset
sigemptyset	sigfillset	sigismember	siglongjmp
signal	sigpending	sigprocmask	sigsuspend
sleep	snprintf	sprintf	srand
srand48	srandom	sscanf	ssignal
stat	stpcpy	stpncpy	strcasecmp
strcat	strchr	strcmp	strcoll
strcpy	strcspn	strdup	strerror
strerror_r	strfmon	strfry	strftime
strlen	strncasecmp	strncat	strncmp
strncpy	strndup	strnlen	strpbrk

[1] see Section 4.3.5, page 36.

strptime	strrchr	strsep	strsignal
strspn	strstr	strtod	strtof
strtok	strtok_r	strtol	strtold
strtoll	strtoq	strtoul	strtoull
strtouq	strverscmp	strxfrm	svcfd_create
swab	swprintf	symlink	syscall
sysconf	tdelete	telldir	textdomain
tfind	time	timegm	timelocal
timezone	tmpfile	toascii	tolower
toupper	towctrans	towlower	towupper
truncate	tsearch	ttyslot	twalk
tzname	tzset	umask	umount
uname	ungetc	ungetwc	unlink
unsetenv	usleep	utime	vasprintf
vdprintf	verr	verrx	versionsort
vfork	vfprintf	vfscanf	vfwprintf
vprintf	vscanf	vsnprintf	vsprintf
vsscanf	vswprintf	vwarn	vwarnx
vwprintf	warn	warnx	wcpcpy
wcpncpy	wcrtomb	wcscasecmp	wcscat
wcschr	wcscmp	wcscpy	wcscspn
wcsdup	wcslen	wcsncasecmp	wcsncat
wcsncmp	wcsncpy	wcsnlen	wcsnrtombs
wcspbrk	wcsrchr	wcsrtombs	wcsspn
wcsstr	wcstok	wcstombs	wcswidth
wctob	wctomb	wctrans	wctype
wcwidth	wmemchr	wmemcmp	wmemcpy
wmemmove	wmemset	wprintf	write
writev	xdecrypt	xencrypt	

glibc Functions Supported in Catamount [B]

The Catamount port of glibc supports the functions listed in Table 10. For further information, see the man pages.

Note: Some `fcntl()` commands are not supported for applications that use Lustre. The supported commands are:

- `F_GETFL`
- `F_SETFL`
- `F_GETLK`
- `F_SETLK`
- `F_SETLKW64`
- `F_SETLKW`
- `F_SETLK64`

The Cray XT series system supports two implementations of `malloc()` for compute nodes running Catamount: Catamount `malloc` and GNU `malloc`. If your code makes generous use of `malloc()`, `alloc()`, `realloc()`, or automatic arrays, you may notice improvements in scaling by loading the GNU `malloc` module and relinking.

To use GNU `malloc`, load the `gmalloc` module:

```
% module load gmalloc
```

Entry points in `libgmalloc.a` (GNU `malloc`) are referenced before those in `libc.a` (Catamount `malloc`).

Table 10. Supported glibc Functions for Catamount

a64l	abort	abs	access
addmntent	alarm	alphasort	argz_add
argz_add_sep	argz_append	argz_count	argz_create
argz_create_sep	argz_delete	argz_extract	argz_insert
argz_next	argz_replace	argz_stringify	asctime

asctime_r	asprintf	atexit	atof
atoi	atol	atoll	basename
bcmp	bcopy	bind_textdomain_codeset	bindtextdomain
bsearch	btowc	bzero	calloc
catclose	catgets	catopen	cbc_crypt
chdir	chmod	chown	clearenv
clearerr	clearerr_unlocked	close	closedir
confstr	copysign	copysignf	copysignl
creat	ctime	ctime_r	daemon
daylight	dcgettext	dcngettext	des_setparity
dgettext	difftime	dirfd	dirname
div	dngettext	dprintf	drand48
dup	dup2	dysize	ecb_crypt
ecvt	ecvt_r	endfsent	endmntent
endttyent	endusershell	envz_add	envz_entry
envz_get	envz_merge	envz_remove	envz_strip
erand48	err	errx	exit
fchmod	fchown	fclose	fcloseall
fcntl	fcvt	fcvt_r	fdatasync
fdopen	feof	feof_unlocked	ferror
ferror_unlocked	fflush	fflush_unlocked	ffs
ffsl	ffsll	fgetc	fgetc_unlocked
fgetgrent	fgetpos	fgetpwent	fgets
fgets_unlocked	fgetwc	fgetwc_unlocked	fgetws
fgetws_unlocked	fileno	fileno_unlocked	finite
flockfile	fnmatch	fopen	fprintf
fputc	fputc_unlocked	fputs	fputs_unlocked
fputwc	fputwc_unlocked	fputws	fputws_unlocked
fread	fread_unlocked	free	freopen
frexp	fscanf	fseek	fseeko

fsetpos
fstat
fsync
ftell
ftello
ftime
ftok
ftruncate
ftrylockfile
funlockfile
fwide
fwprintf
fwrite
fwrite_unlocked
gcvt
get_current_dir_name
getc
getc_unlocked
getchar
getchar_unlocked
getcwd
getdate
getdate_r
getdelim
getdirentries
getdomainname
getegid
getenv
geteuid
getfsent
getfsfile
getfsspec
getgid
gethostname
getline
getlogin
getlogin_r
getmntent
getopt
getopt_long
getopt_long_only
getpagesize
getpass
getpid
getrlimit
getrusage
gettext
gettimeofday
getttyent
getttynam
getuid
getusershell
getw
getwc
getwc_unlocked
getwchar
getwchar_unlocked
gmtime
gmtime_r
gsignal
hasmntopt
hcreate
hcreate_r
hdestroy
hsearch
iconv
iconv_close
iconv_open
imaxabs
index
initstate
insque
ioctl
isalnum
isalpha
isascii
isblank
iscntrl
isdigit
isgraph
isinf
islower
isnan
isprint
ispunct
isspace
isupper
iswalnum
iswalpha
iswblank
iswcntrl
iswctype
iswdigit
iswgraph
iswlower
iswprint
iswpunct
iswspace
iswupper
iswxdigit
isxdigit
jrand48
kill
l64a
labs
lcong48
ldexp
lfind
link
llabs
localeconv
localtime
localtime_r
lockf
longjmp
lrand48
lsearch
lseek
lstat
malloc

mblen	mbrlen	mbrtowc	mbsinit
mbsnrtowcs	mbsrtowcs	mbstowcs	mbtowc
memccpy	memchr	memcmp	memcpy
memfrob	memmem	memmove	memrchr
memset	mkdir	mkdtemp	mknod
mkstemp	mktime	modf	modff
modfl	mrand48	nanosleep	ngettext
nl_langinfo	nrand48	on_exit	open
opendir	passwd2des	pclose	perror
pread	printf	psignal	putc
putc_unlocked	putchar	putchar_unlocked	putenv
putpwent	puts	putw	putwc
putwc_unlocked	putwchar	putwchar_unlocked	pwrite
qecvt	qecvt_r	qfcvt	qfcvt_r
qgcvt	qsort	raise	rand
random	re_comp	re_exec	read
readdir	readlink	readv	realloc
realpath	regcomp	regerror	regexec
regfree	registerrpc	remove	remque
rename	rewind	rewinddir	rindex
rmdir	scandir	scanf	seed48
seekdir	setbuf	setbuffer	setegid
setenv	seteuid	setfsent	setgid
setitimer	setjmp	setlinebuf	setlocale
setlogmask	setmntent	setrlimit	setstate
setttyent	setuid	setusershell	setvbuf
sigaction	sigaction[1]	sigaddset	sigdelset
sigemptyset	sigfillset	sigismember	siglongjmp

[1] see Section 4.3.5, page 36.

signal
sigpending
sigprocmask
sigsuspend
sleep
snprintf
sprintf
srand
srand48
srandom
sscanf
ssignal
stat
stpcpy
stpncpy
strcasecmp
strcat
strchr
strcmp
strcoll
strcpy
strcspn
strdup
strerror
strerror_r
strfmon
strfry
strftime
strlen
strncasecmp
strncat
strncmp
strncpy
strndup
strnlen
strpbrk
strptime
strrchr
strsep
strsignal
strspn
strstr
strtod
strtof
strtok
strtok_r
strtol
strtold
strtoll
strtoq
strtoul
strtoull
strtouq
strverscmp
strxfrm
svcfd_create
swab
swprintf
symlink
syscall
sysconf
tdelete
telldir
textdomain
tfind
time
timegm
timelocal
timezone
tmpfile
toascii
tolower
toupper
towctrans
towlower
towupper
truncate
tsearch
ttyslot
twalk
tzname
tzset
umask
umount
uname
ungetc
ungetwc
unlink
unsetenv
usleep
utime
vasprintf
vdprintf
verr
verrx
versionsort
vfork
vfprintf
vfscanf
vfwprintf
vprintf
vscanf
vsnprintf
vsprintf
vsscanf
vswprintf
vwarn
vwarnx
vwprintf
warn
warnx
wcpcpy
wcpncpy
wcrtomb
wcscasecmp
wcscat
wcschr
wcscmp
wcscpy
wcscspn

wcsdup	wcslen	wcsncasecmp	wcsncat
wcsncmp	wcsncpy	wcsnlen	wcsnrtombs
wcspbrk	wcsrchr	wcsrtombs	wcsspn
wcsstr	wcstok	wcstombs	wcswidth
wctob	wctomb	wctrans	wctype
wcwidth	wmemchr	wmemcmp	wmemcpy
wmemmove	wmemset	wprintf	write
writev	xdecrypt	xencrypt	

PAPI Hardware Counter Presets [C]

The following table describes the hardware counter presets that are available on the Cray XT series system. Use these presets to construct an event set as described in Section 11.1.2, page 84.

Table 11. PAPI Presets

Name	Supported on Cray XT series	Derived from multiple counters?	Description
PAPI_L1_DCM	Yes	No	Level 1 data cache misses
PAPI_L1_ICM	Yes	No	Level 1 instruction cache misses
PAPI_L2_DCM	Yes	No	Level 2 data cache misses
PAPI_L2_ICM	Yes	No	Level 2 instruction cache misses
PAPI_L3_DCM	No	No	Level 3 data cache misses
PAPI_L3_ICM	No	No	Level 3 instruction cache misses
PAPI_L1_TCM	Yes	Yes	Level 1 cache misses
PAPI_L2_TCM	Yes	No	Level 2 cache misses
PAPI_L3_TCM	No	No	Level 3 cache misses
PAPI_CA_SNP	No	No	Requests for a snoop
PAPI_CA_SHR	No	No	Requests for exclusive access to shared cache line
PAPI_CA_CLN	No	No	Requests for exclusive access to clean cache line
PAPI_CA_INV	No	No	Requests for cache line invalidation
PAPI_CA_ITV	No	No	Requests for cache line intervention
PAPI_L3_LDM	No	No	Level 3 load misses
PAPI_L3_STM	No	No	Level 3 store misses
PAPI_BRU_IDL	No	No	Cycles branch units are idle

Name	Supported on Cray XT series	Derived from multiple counters?	Description
PAPI_FXU_IDL	No	No	Cycles integer units are idle
PAPI_FPU_IDL	No	No	Cycles floating-point units are idle
PAPI_LSU_IDL	No	No	Cycles load / store units are idle
PAPI_TLB_DM	Yes	No	Data translation lookaside buffer misses
PAPI_TLB_IM	Yes	No	Instruction translation lookaside buffer misses
PAPI_TLB_TL	Yes	Yes	Total translation lookaside buffer misses
PAPI_L1_LDM	Yes	No	Level 1 load misses
PAPI_L1_STM	Yes	No	Level 1 store misses
PAPI_L2_LDM	Yes	No	Level 2 load misses
PAPI_L2_STM	Yes	No	Level 2 store misses
PAPI_BTAC_M	No	No	Branch target address cache misses
PAPI_PRF_DM	No	No	Data prefetch cache misses
PAPI_L3_DCH	No	No	Level 3 data cache hits
PAPI_TLB_SD	No	No	Translation lookaside buffer shootdowns
PAPI_CSR_FAL	No	No	Failed store conditional instructions
PAPI_CSR_SUC	No	No	Successful store conditional instructions
PAPI_CSR_TOT	No	No	Total store conditional instructions
PAPI_MEM_SCY	Yes	No	Cycles Stalled Waiting for memory accesses
PAPI_MEM_RCY	No	No	Cycles Stalled Waiting for memory reads

Name	Supported on Cray XT series	Derived from multiple counters?	Description
PAPI_MEM_WCY	No	No	Cycles Stalled Waiting for memory writes
PAPI_STL_ICY	Yes	No	Cycles with no instruction issue
PAPI_FUL_ICY	No	No	Cycles with maximum instruction issue
PAPI_STL_CCY	No	No	Cycles with no instructions completed
PAPI_FUL_CCY	No	No	Cycles with maximum instructions completed
PAPI_HW_INT	Yes	No	Hardware interrupts
PAPI_BR_UCN	Yes	No	Unconditional branch instructions
PAPI_BR_CN	Yes	No	Conditional branch instructions
PAPI_BR_TKN	Yes	No	Conditional branch instructions taken
PAPI_BR_NTK	Yes	Yes	Conditional branch instructions not taken
PAPI_BR_MSP	Yes	No	Conditional branch instructions mispredicted
PAPI_BR_PRC	Yes	Yes	Conditional branch instructions correctly predicted
PAPI_FMA_INS	No	No	FMA instructions completed
PAPI_TOT_IIS	No	No	Instructions issued
PAPI_TOT_INS	Yes	No	Instructions completed
PAPI_INT_INS	No	No	Integer instructions
PAPI_FP_INS	Yes	No	Floating-point instructions
PAPI_LD_INS	No	No	Load instructions
PAPI_SR_INS	No	No	Store instructions
PAPI_BR_INS	Yes	No	Branch instructions
PAPI_VEC_INS	Yes	No	Vector / SIMD instructions

Name	Supported on Cray XT series	Derived from multiple counters?	Description
PAPI_FLOPS	Yes	Yes	Floating-point instructions per second
PAPI_RES_STL	Yes	No	Cycles stalled on any resource
PAPI_FP_STAL	Yes	No	Cycles in the floating-point unit(s) are stalled
PAPI_TOT_CYC	Yes	No	Total cycles
PAPI_IPS	Yes	Yes	Instructions per second
PAPI_LST_INS	No	No	Load/store instructions completed
PAPI_SYC_INS	No	No	Synchronization instructions completed
PAPI_L1_DCH	Yes	Yes	Level 1 data cache hits
PAPI_L2_DCH	Yes	No	Level 2 data cache hits
PAPI_L1_DCA	Yes	No	Level 1 data cache accesses
PAPI_L2_DCA	Yes	No	Level 2 data cache accesses
PAPI_L3_DCA	No	No	Level 3 data cache accesses
PAPI_L1_DCR	No	No	Level 1 data cache reads
PAPI_L2_DCR	Yes	No	Level 2 data cache reads
PAPI_L3_DCR	No	No	Level 3 data cache reads
PAPI_L1_DCW	No	No	Level 1 data cache writes
PAPI_L2_DCW	Yes	No	Level 2 data cache writes
PAPI_L3_DCW	No	No	Level 3 data cache writes
PAPI_L1_ICH	No	No	Level 1 instruction cache hits
PAPI_L2_ICH	No	No	Level 2 instruction cache hits
PAPI_L3_ICH	No	No	Level 3 instruction cache hits
PAPI_L1_ICA	Yes	No	Level 1 instruction cache accesses
PAPI_L2_ICA	Yes	No	Level 2 instruction cache accesses
PAPI_L3_ICA	No	No	Level 3 instruction cache accesses

Name	Supported on Cray XT series	Derived from multiple counters?	Description
PAPI_L1_ICR	Yes	No	Level 1 instruction cache reads
PAPI_L2_ICR	No	No	Level 2 instruction cache reads
PAPI_L3_ICR	No	No	Level 3 instruction cache reads
PAPI_L1_ICW	No	No	Level 1 instruction cache writes
PAPI_L2_ICW	No	No	Level 2 instruction cache writes
PAPI_L3_ICW	No	No	Level 3 instruction cache writes
PAPI_L1_TCH	No	No	Level 1 total cache hits
PAPI_L2_TCH	No	No	Level 2 total cache hits
PAPI_L3_TCH	No	No	Level 3 total cache hits
PAPI_L1_TCA	Yes	Yes	Level 1 total cache accesses
PAPI_L2_TCA	No	No	Level 2 total cache accesses
PAPI_L3_TCA	No	No	Level 3 total cache accesses
PAPI_L1_TCR	No	No	Level 1 total cache reads
PAPI_L2_TCR	No	No	Level 2 total cache reads
PAPI_L3_TCR	No	No	Level 3 total cache reads
PAPI_L1_TCW	No	No	Level 1 total cache writes
PAPI_L2_TCW	No	No	Level 2 total cache writes
PAPI_L3_TCW	No	No	Level 3 total cache writes
PAPI_FML_INS	Yes	No	Floating-point multiply instructions
PAPI_FAD_INS	Yes	No	Floating-point add instructions
PAPI_FDV_INS	No	No	Floating-point divide instructions
PAPI_FSQ_INS	No	No	Floating-point square root instructions
PAPI_FNV_INS	Yes	Yes	Floating-point inverse instructions. This event is available only if you compile with the -DDEBUG flag.

MPI Error Messages [D]

Table 12 lists the MPI error messages you may encounter and suggested workarounds.

Table 12. MPI Error Messages

Message	Description	Workaround
`Segmentation fault in MPID_Init()`	The application is using all the memory on the node and not leaving enough for MPI's internal data structures and buffers.	Reduce the amount of memory used for MPI buffering by setting the environment variable `MPICH_UNEX_BUFFER_SIZE` to something greater than 60 MB. If the application uses scalable data distribution, run at higher process counts.
`MPIDI_PortalsU_Request_PUPE(323): exhausted unexpected receive queue buffering increase via env. var. MPICH_UNEX_BUFFER_SIZE`	The application is sending too many short, unexpected messages to a particular receiver.	Increase the amount of memory for MPI buffering using the `MPICH_UNEX_BUFFER_SIZE` environment variable or decrease the short message threshold using the `MPICH_MAX_SHORT_MSG_SIZE` variable (default is 128 KB). The default for `MPICH_UNEX_BUFFER_SIZE` is 60,000,000 bytes. The `MPICH_UNEX_BUFFER_SIZE` environment variable specifies the entire amount of buffer space for short unexpected messages.

Message	Description	Workaround
pe_rank MPIDI_Portals_Progress: dropped event on unexpected receive queue, increase *pe_rank* queue size by setting the environment variable MPICH_PTL_UNEX_EVENTS	You have used up all the space allocated for event queue entries associated with the unexpected messages queue. The default size is 20,480 bytes.	You can increase the size of the unexpected messages event queue by setting the environment variable MPICH_PTL_UNEX_EVENTS to a value higher than 20,480 bytes.
pe_rank MPIDI_Portals_Progress: dropped event on "other" queue,increase *pe_rank* queue size by setting the environment variable MPICH_PTL_OTHER_EVENTS	You have used up all the space allocated for the event queue entries associated with the "other" queue. This can happen if the application is posting many non-blocking sends of large messages, or many MPI-2 RMA operations are posted in a single epoch. The default size is 2048 bytes.	You can increase the size of the queue by setting the environment variable MPICH_PTL_OTHER_EVENTS to a value higher than 2048 bytes.

ALPS Error Messages [E]

This appendix documents common ALPS error messages. It is possible for you to see many more messages than those documented here. Other messages are generated only if a system error occurs. For all ALPS messages not described here, see your system administrator.

These messages are generated by the placement scheduler during application placement and are forwarded to the user through `aprun`.

Messages that begin with [NID *nnn*] come from the application shepherds on the compute nodes and are prefixed with a node ID (NID) to indicate which compute node sent the message. When general application failures occur, typically only one message appears from an arbitrary NID assigned to the application. This is done to prevent flooding the user with possibly thousands of identical messages if the application fails globally.

Table 13. ALPS Error Messages

Error	Description
`no XT nodes are configured up`	A request for the named type of compute node cannot be satisfied because there are no nodes of that type currently available.
`memory request exceeds 1048575 megabytes`	The `aprun -m` value exceeds the indicated amount. This is probably a mistake in units by the user because the value far exceeds any compute node memory size possible to install.
`Request exceeds max [CPUs | memory | nodes]` `In user NIDs request exceeds max [CPUs | memory | nodes]`	The allocation request requires more of the named resource than the configuration can deliver at this time. The second message will appear instead of the first if the user has specified the NIDs using the `aprun -L` option.
`At least one command's user NID list is short`	If the `aprun -L` option is used, the NID list must have at least as many NID values as the number of nodes the application requires.
`nid` *`NNN`* `appears more than once in user's nid list`	The user has specified an NID list, but the list has at least one duplicate NID.
`[NID` *`nnn`*`] Apid` *`NNNN`* `/proc readdir timeout alarm occurred. Application aborted.`	A problem on the node prevented the shepherd responsible for the application to read information from `/proc` as it must. Report this to the system administrator.

Error	Description
[NID *nnn*] Apid *NNNN*: cannot execute: *reason*	A large number of reasons can appear, but the most likely is exec failed, which usually means the a.out file is corrupted or is the wrong instruction set to run on this compute node.
[NID *nnn*] Apid *NNNN* killed. Received node failed or unavailable event for nid *nnn*	The system monitoring software has detected an unrecoverable error on the named NID. Notification has been delivered to this NID for handling. The application must be killed because one or more of the compute nodes on which it is running have failed.
aprun: Exiting due to errors. Launch aborted	Typically, this is the final message from aprun before it terminates when an error has been detected. More detailed messages should precede this one.
aprun: Apid *NNNN* close of the compute node connection [before | after] app startup barrier	The compute node to which aprun is connected has dropped its socket connection to aprun without warning. This usually means the application or a compute node has failed in some way that prevents normal error messages from being created or delivered to aprun.
aprun: Application *NNNN* exit codes: *one to four values* aprun: Application *NNNN* exit signals: *one to four values*	If an application terminates with nonzero exit codes or has internally generated a signal (such as a memory address error), the first four of the values detected are reported with these messages. Both messages will appear if both nonzero exit codes and signals have occurred in the application.
aprun: Application *NNNN* resources: utime *uuu*, stime *sss*	When the application terminates the accumulated user time (utime) and system time (stime) are forwarded to aprun and reported with this message.

yod Error Messages [F]

Table 14 describes yod error messages.

Table 14. yod Error Messages

Error	Number	Description
ERR_NO_MEMORY	1	Out of memory in yod.
ERR_USAGE	2	Command-line usage error.
ERR_HOST_INIT	3	Error in host_cmd_init due to out of memory or portals. yod internal initialization failed.
ERR_MESH_ALLOC	8	Call to mesh_alloc failed. Error during mesh initialization.
ERR_LOAD	9	Load error. Cannot load program.
ERR_ABORT	10	User aborted yod. yod was aborted during load of program.
LD_ERR_SEND	10	Error while sending data to children in fan-out tree.
LD_ERR_NO_HEAP	10	Error allocating heap memory on node.
LD_ERR_TARGET_LENGTH	10	Target supplied location too small for message to be sent.
ERR_LOAD_FILE	13	Load-file error. Error in use of heterogeneous load file.
ERR_YOD_USAGE	14	General yod usage error.
ERR_KILL	23	Application was killed. yod got killed after load.
ERR_TARGET	26	Invalid target option; valid targets are linux and catamount.
ERR_TIME_LIMIT	27	yod time limit expired.
ERR_PREMATURE_EXIT	28	yod received CMD_EXIT too soon. A process exited prematurely.
ERR_ALARM	29	Load time-out. Alarm signal.
ERR_RCA	30	RCA register failed.
LD_ERR_ABORTED	100	Aborted load.

Error	Number	Description
LD_ERR_START	100	First load error.
LD_ERR_NUMNODES	101	Number of nodes was outside of range allowed.
LD_ERR_INTERNAL	102	Internal error.
PCT LD_ERR_CONTROL_PORTAL	103	Error on control portal.
LD_ERR_TARGET_RANK	105	Rank of requesting node is out of expected range.
LD_ERR_TARGET_PORTAL	106	Target portal number is out of expected range.
LD_ERR_PULL	108	Error while pulling data from parent in fan-out tree.
LD_ERR_VERSION	110	Version mismatch.
LD_ERR_NODE_TIMEOUT	111	Time-out while communicating with node.
LD_ERR_PORTALS_UID	112	Portals UID mismatch.
LD_ERR_PROTOCOL_ERROR	113	General load-protocol error.
LD_ERR_BAD_PCT_MSG_TYPE	114	Unexpected message type.
LD_ERR_EXEC_LOAD	115	Error loading executable file.
LD_ERR_WRONG_NID	116	Received response from wrong node ID.
LD_ERR_WRONG_RECV_LENGTH	117	Received load with wrong length.
LD_ERR_PCT_EXIT	118	PCT exited during load.
LD_ERR_NIDPID	119	Node ID map was built or distributed incorrectly.
ERROR_PCT_FAULT	120	PCT fault.
ERROR_SET_CACHE	121	PCT failed to initialize processor.
ERROR_INIT_REGION	122	PCT failed to initialize memory region.
ERROR_APP_TIMER	123	Application Timer Error.
ERROR_NO_MEM	124	Out of memory on node.
ERROR_NO_MEM_FOR_BSS	125	Text size is too big.
ERROR_NO_MEM_FOR_HEAP	126	Not enough memory for heap on node.
ERROR_NO_MEM_FOR_PROCESS	127	Not enough memory for process.
ERROR_HEAP_SIZE_TOO_SMALL	128	Heap size is too small on node.
ERROR_NO_SMP	129	Catamount virtual node mode is unavailable.
ERROR_VA_OVERLAP	130	Virtual addresses overlap kernel/PCT addresses.

Error	Number	Description
ERROR_PRIORITY	131	PCT could not set processor priority.
ERROR_PORTALS	132	Portals Error.
ERROR_BAD_ELF_FILE	133	Bad ELF file.
ERROR_ELF_DYNAMIC_LOAD	134	No dynamic load support for ELF files.
ERROR_ELF_GENERIC	135	ELF file error.
ERROR_INVALID_TARGET	136	Invalid target.
ERROR_MSG_RCV_CACHE_OVERFLOW	137	Overflow in message received cache.
ERROR_TOO_MANY_PARAMS	138	Too many parameters passed to application
ERROR_TOO_MANY_PORTALS	139	Too many portals were allocated.
ERROR_TOO_MANY_PROCS	140	Too many processes.

Glossary

Catamount

The operating system kernel developed by Sandia National Laboratories and implemented to run on Cray XT series compute nodes. See also *compute node*.

Catamount Virtual Node (CVN)

The Catamount kernel enhanced to run on dual-core Cray XT series compute nodes.

CNL

CNL is a Cray XT series compute node operating system. CNL provides a set of supported system calls. CNL provides many of the operating system functions available through the service nodes, although some functionality has been removed to improve performance and reduce memory usage by the system.

compute node

Runs a kernel and performs only computation. System services cannot run on compute nodes. See also *node*; *service node*.

compute processor allocator (CPA)

A program that coordinates with yod to allocate processing elements.

CrayDoc

Cray's documentation system for accessing and searching Cray books, man pages, and glossary terms from a web browser.

deferred implementation

The label used to introduce information about a feature that will not be implemented until a later release.

dual-core processor

A processor that combines two independent execution engines ("cores"), each with its own cache and cache controller, on a single chip.

login node

The service node that provides a user interface and services for compiling and running applications.

Modules

A package on a Cray system that allows you to dynamically modify your user environment by using module files. (This term is not related to the module statement of the Fortran language; it is related to setting up the Cray system environment.) The user interface to this package is the `module` command, which provides a number of capabilities to the user, including loading a module file, unloading a module file, listing which module files are loaded, determining which module files are available, and others.

node

For UNICOS/lc systems, the logical group of processor(s), memory, and network components acting as a network end point on the system interconnection network.

node ID

A decimal number used to reference each individual node. The node ID (NID) can be mapped to a physical location.

service node

A node that performs support functions for applications and system services. Service nodes run SUSE LINUX and perform specialized functions. There are six types of predefined service nodes: login, IO, network, boot, database, and syslog.

system interconnection network

The high-speed network that handles all node-to-node data transfers.

UNICOS/lc

The operating system for Cray XT series systems.

Index